НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій Стіренко

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Система управління контентом»

Виконав:
студент IV курсу, групи ІП-62
Дубинський Олег Ігорович _____

Керівник:
старший викладач, кандидат технічних наук
Сімоненко Валерій Павлович _____

Консультант з нормокнтролю:
професор, доктор технічних наук
Сімоненко Валерій Павлович _____

Рецензент:
_____
_____ _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій Стіренко
«___»_____20__ р.

ЗАВДАННЯ
на дипломний проєкт студенту
Дубинському Олегу Ігоровичу

1. Тема проєкту «Система управління контентом», керівник проєкту Сімоненко Валерій Павлович, доктор технічних наук, професор, затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту: теоретичні відомості, технічна документація, система управління контентом.

4. Зміст пояснювальної записки: огляд існуючих систем управління контентом, проєктування системи управління контентом, реалізація системи управління контентом, огляд створеної системи управління контентом.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо): діаграма класів, структурна схема системи, діаграма прецедентів.

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Сімоненко В. П., професор		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Затвердження теми роботи	10.12.2019	
2	Вивчення та аналіз завдання	10.12.2019 - 02.02.2020	
3	Розробка архітектури та загальної структури системи	02.02.2020 - 17.03.2020	
4	Розробка структур окремих підсистем	17.03.2020 - 12.04.2020	
5	Програмна реалізація системи	12.04.2020 - 05.05.2020	
6	Виправлення помилок	05.05.2020 - 09.05.2020	
7	Оформлення пояснювальної записки	09.05.2020 - 06.06.2020	
8	Передзахист		
9	Захист		

Студент Дубинський Олег Ігорович

Керівник Сімоненко Валерій Павлович

АНОТАЦІЯ

У даній роботі детально розглянуто існуючі класичні та headless системи управління контентом, а також їх принцип роботи. На основі існуючих рішень було запропоновано новий підхід написання headless системи управління контентом. СКВ була розділена на клієнтську та серверну частини, які взаємодіють за допомогою певного протоколу. В рамках роботи було розроблено систему управління контентом, що отримала чесноти існуючих рішень та немає їх вад.

ABSTRACT

This work analyses in detail existing classic and headless content management systems and their work principle. A new way of creating headless content management systems was proposed on the basis of existing solutions. CMS was spitted into client and server parts that interact with each other using specific protocol. Content management system that has pros of existing solutions and is free from their cons was implemented in terms of this work.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1.	А4		Завдання на дипломний проєкт	2	
2.	А4	ІАЛЦ.467200.001 ВП	Відомість на дипломний проєкт	1	
3.	А4	ІАЛЦ.467200.002 ТЗ	Технічне завдання	3	
4.	А4	ІАЛЦ.467200.003 ПЗ	Пояснювальна записка	68	
5.	А1	ІАЛЦ.467200.004 Д1	Схема структурна	1	
6.	А1	ІАЛЦ.467200.005 Д2	Діаграма класів	1	
7.	А1	ІАЛЦ.467200.006 Д3	Діаграма прецендентів	1	
8.	А4	ІАЛЦ.467200.007 Д4	Лістинг програми	41	

				ІАЛЦ.467200.001 ВП		
	ПІБ	Підп.	Дата			
Розробн.	Дубинський О.І.			Відомість дипломного проєкту	Лист	Листів
Керівн.	Сімоненко В.П.				1	1
					КПІ ім. Ігоря Сікорського Каф. ОТ Гр. ІП-62	
Н/контр.	Сімоненко В.П.					
Зав.каф.	Стіренко С.Г.					

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЕКТУ

на тему: «Система управління контентом»

Київ – 2020

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання стосується розробки системи управління контенту, яка не відповідає за представлення даних, а дає змогу управляти власне контентом.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврської дипломної роботи, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут» імені Ігоря Сікорського.

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення Headless системи управління контентом.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література, монографії, публікації в періодичних виданнях та мережі Інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

Система, що розробляється, повинна:

1) бути розподіленим веб-додатком, що складається з адміністративної панелі та серверу взаємодія між якими відбувається за допомогою розробленого протоколу СКВ.

2) бути легко розширюваною, не залежною від баз даних, має надавати більший функціонал ніж існуючі рішення.
3) надавати можливість управляти контентом, станом системи та переглядати документацію в зручній адміністративній панелі.

5.2. Вимоги до інструментального програмного забезпечення

- середовище з встановленим збірником sbt версії 1.3.10 та AdoptOpenJdk версії 1.8;

5.3. Вимоги до апаратної частини обчислювальної системи

- процесор з тактовою частотою не менше ніж 800 КГц;
- оперативна пам’ять об’ємом не менше ніж 700 Мб;
- жорсткий диск об’ємом не менше ніж 200 Мб.

Пояснювальна записка
до дипломного проєкту
на тему: «Система управління контентом»

Київ – 2020 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

CMS – Content Management System

CRUD – Create Read Update Delete

API – Application Programming Interface

БД – База Даних

СКБД – Система Керування Базами Даних

ІС – Інформаційна Система

CLI – Command Line Interface

WYSIWYG – What You See Is What You Get

SDK – Software development kit

SPA – Single page application

ВСТУП

В наш час ІС стали невід'ємною частиною будь-якого бізнесу в різноманітних сферах діяльності: онлайн-магазини, запис до лікаря на прийом, бронювання стола в ресторанах. Можливостям ІС і потребам бізнесів немає кінця, вигадують все більше нових інноваційних підходів використання ІС в веденні бізнесу і об'єднує всі ці потреби те що вони оперують над певним контентом, що зберігається на сервері та відображається клієнтською частиною. Спочатку все було примітивно та потребувало чималих коштів, щоб бізнес почав заробляти гроші онлайн, проте інженери почали бачити однотипні речі, які можна було б автоматизувати, а не робити все, як по шаблону. Саме так і почали з'являтись системи управління контентом. На сьогоднішній день вже існує чимало рішень і всі вони користуються попитом, хтось більшою мірою, а хтось меншою, проте кожне рішення знаходить своїх користувачів. CMS системи дуже різняться потужністю, функціоналом, прив'язкою до мови програмування, зручністю розробки та наскільки важлива участь розробника при роботі з такою системою. Досить популярними є системи, що дають можливість створювати ІС людині далекій від програмування шляхом звичайних маніпуляцій в адміністративній панелі, проте такі рішення досить часто обмеженні з «коробки» і потребую додаткових маніпуляцій та плагінів, тому серйозні рішення зазвичай мають окремий шар відображення. Проте рутинні, шаблонні операції при розробці сервера для такого клієнта все ще присутні і рішень на ринку які націлені лише на серверну частину досить мало, а ті що є часто мають чимало обмежень, як для клієнтів так і для розробників, в основному прив'язані до конкретної мови програмування і не підходять на пару з мікросервісною архітектурою. Саме тому була розроблена CMS система представлена в даній дипломній роботі

Основною відмінністю реалізованої системи є висока швидкодія та гарна утилізація ресурсів системи, можливість розширювати систему для своїх потреб, можливість використовувати будь-яке сховище для контенту, гнучкий адміністративний веб-інтерфейс не прив’язаний до конкретної мови програмування, а також з купою додаткового функціоналу, що є тільки плюсом для такого роду системи.

Користувач матиме можливість створення, редагування, видалення, пошуку контентом, надавати ролі доступу до контенту, слідкувати за станом «здоров’я» серверів, перегляди документацію серверів, а також буде наявно WEB API, що можна використовувати при розробці клієнтської частини.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ СИСТЕМ УПРАВЛІННЯ КОНТЕНТОМ

1.1 Опис предметного середовища

Веб-додаток – розподілений застосунок, що має окремого клієнта, будь то браузер, мобільний додаток чи додаток для персонального комп'ютера, а сервером - веб-сервер, при цьому їх комунікація відбувається через Інтернет за допомогою мережевих протоколів, таких як HTTP, HTTP2, UDP, TCP, FTP та інші за потреби.

Особливістю такого підходу є те, що всі дані отримані від клієнта зберігаються та оброблюються на сервері таким чином вся логіка застосунку інкапсулюється і приховується від користувача, який лише надсилає та отримує дані в певному форматі, а браузер при цьому виступає в якості клієнта та інтерфейса для користувача. Величезним плюсом таких застосунків є те, що користувач не повинен завантажувати додаткові застосунки, щоб отримати доступ до веб-додатку, йому достатньо мати гарний інтернет та браузер, а решта потрібних для веб-додатку речей зазвичай або викачується прямо при рендерингу сторінки або весь рендеринг відбувається на сервері повністю і користувач отримує готову HTML сторінку – це так званий SSR. З вказаного вище виходить наступні переваги такого підходу – програми мають невеликий обсяг, вони компактні, швидкі, прості в створенні та використанні, а також дуже інтерактивні.

Веб-додатки можуть мати різні клієнтські частини, що спілкуються з одним і тим же веб-сервером, проте наразі найпопулярнішим і найлегшим дял створення є саме браузерний клієнт і для його створення використовуються такі технології, як HTML та CSS. Якщо перший використовується для розмітки сторінки та елементів, що знаходяться на ній, то другий формує зовнішній

вигляд сторінки і його використовують для створення графічних ефектів, анімацій та іншої стилізації сторінки.

В минулому крім цих технологій нічого не було і найпопулярнішим рішенням на ринку були статичні сайти та сайти з SSR, в яких логіка відображення мішалась з серверною і клієнт отримував готову HTML сторінку такий підхід називається товстий клієнт, адже чималу роботу по відображеню виконує сервер, хоча він міг витратити це час на більш «корисні» задачі.

Веб-додатки створенні за допомогою лише HTML та CSS є статичними і не надають жодних варіантів інтерактивності користувачу, а також при потребі створення не статичного сайту, а повноцінного веб-додатка при такому підході доводиться виконувати генерацію сторінок динамічно на стороні сервера. Саме тому в браузері існує рішення для цього і назва йому - JavaScript. Це скриптова, динамічно-типізована мова, яку вміє виконувати кожен сучасний браузер і саме її використовують для побудови інтерактивних веб-додатків, що взаємодіють з веб-сервером. Іншим плюсом цього підходу є відокремлення логіки сервера та клієнта, що відкриває простір до створення мікросервісних архітектур, інтеграцій з іншими системами прямо на стороні клієнта без потреби веб-серверу знати про це, економляться ресурси веб-серверу і він більшість часу витрачає на важливу бізнес логіку, а не рендеринг сторінок для клієнта, який в свою чергу просто робить запити задля взаємодії з певною інформацією, що зберігається на сервері.

В наш час вже давно не пишуть на звичайному JavaScript, HTML, CSS стеці технологій, бо це є зовсім непродуктивно і потребує чимало ресурсів, саме тому існують готові рішення-фреймворки так як Angular, Vue, React, Ember, що виступають в ролі автоматизації написання логіки відображення контенту та взаємодії з сервером.

Веб-сервер – це сервер, що приймає запити від клієнтів по мережевому протоколу (HTTP, HTTP2, TCP, UDP та інші) та віддає відповіді в певному форматі. Більшість операцій, що виконує веб-сервер підходять під одну з

категорій CRUD операцій, тобто зазвичай це створення, читання, оновлення та видалення даних. Найпопулярнішими є рішення, що працюють по HTTP протоколі, а найпопулярнішим рішенням для транспортування даних є текстовий формат JSON, бо крім того що дає змогу швидко серілізувати ти десеріалізувати дані довільного формату, його може прочитати людина. Він прийшов на заміну XML формату і досі є лідером в свої області застосування.

Оскільки веб-сервер ніяк не залежить від клієнта і єдина умова для нього це вміння працювати по тому чи іншому мережевому протоколі, то різноманіття способів написання веб-серверів існує чимало для кожної з існуючих мов програмування. Існують як реалізації веб-серверів для конкретних мов, так і клієнтів для взаємодії з іншими серверами, що дає змогу веб-серверам спілкуватись один з одним і таким чином розробники мають ще більший простір для дій. Наразі найпопулярнішими мовами для написання веб-серверів є Java, Python, Scala, C#, Javascript(Node.js) та інші, рішень існує чимало і кожне має свої переваги та недоліки.

1.2 Теоретичні відомості

Система управління контентом – це інформаційна система для організації даних, контентом, розміткою сторінок веб-сайтів, управління цими даним, контролем доступу до них та контролем версій, при цьому це все відбувається автоматизовано і шаблонно, що потребує меншу кількість ресурсів при розробці. Такі системи створили в час, коли розробники почали розуміти, що вони виконують однотипні задачі і вирішили це автоматизувати, деякі рішення настільки автоматизовані, що для роботи з ним не потрібно навичок програмування і достатньо бути впевненим користувачем ПК. Існує чимало таких систем вони бувають як безкоштовні з відкритим кодом та і платні. Різняться вони також за набором можливостей, що вони надають. Проте всі ці рішення об’єднує ідея реалізації шаблонних операцій і надання веб-інтерфейсу для роботи з ними. Тобто такі системи мають дві складові застосунок

управління контентом(CMA) – front-end частина CMS системи та власне сам веб-сервер, що реалізує всі операції, який часто називають CDA(content delivery application). Зазвичай СКВ дуже обмежені і часто розраховані на роботу у певному заздалегіть визначеному середовищі, що накладає ряд обмежень на користувачів таких систем і зменшує їх здатність до розширення, багато з СКВ обходять це обмеження шляхом використання плагінних архітектур, що дозволяє розробникам чи звичайним користувачам доставляти на існуючу СКВ певні розширення, щоб додавати новий функціонал в СКВ.

Спільні властивості:

- Веб-сервер для індексування, змінення, пошуку та отримання даних.
- Управління версіями контенту
- Управління форматом даних
- Веб-застосунок, що взаємодіє з веб-сервером CMS системи для зручної роботи з даними, що включає:
    - Інтуітивний пошук конкретних чи всіх даних, можливість шукати за ключовими словами, за ключами, певними критеріями, датами, авторами та інше
    - Доступність всіх CRUD операцій над контентом
    - Настройка прав доступу користувачів на управління контентом в системі
    - Керування і перегляд історією змін даних
    - Опціонально відображається інформація про стан веб-сервера, документацію наявних веб-інтерфейсів
    - Авторизація та реєстрація в CMS системі
- Функціональність для створення та редагування веб-сторінок, що будуть включені в додаток.

Застосування СКВ систем обширне тому існує велика кількість їх різновидів: для управління веб сайтами(WCMS), транзакційні, інтегровані,

улектронні бібліотеки або Digital Asset Management, для управління документацією, освітні, платформні, корпоративні та інші. Проте найвідмінніша риса сучасних СКВ – це наявність можливості редагувати та створювати сторінки прямо в цій системі. В залежності від наявності цієї риси СКВ поділяються на монолітні та безголові або як їх називають Headless CMS. Традиційні CMS є монолітними, тобто зв'язаними, що означає, що рівень відображення(CMA) та сервер(CDA) пов'язані між собою і існують, як один цілий монолітний додаток, що робить їх нероздільними один від одного. При цьому Headless - система управління контентом, що немає CMA, тобто має лише веб-сервер, що надає можливість доступатися до даних через REST API та все ще надає веб-інтерфейс для управління цими даними, тобто має всі особливості СКВ, проте немає функціональності для створення та редагування сторінок – їх потрібно буде писати власноруч і цей клієнт має змогу взаємодіяти з даними через наданий СКВ веб-інтерфейс. Це створює можливість розділити логіку представлення та роботи з даними.

Переваги одного з рішень виходить з недоліків іншого і навпаки, тому буде доцільно навести плюси та мінуси headless підходу:

- Багатоканальність – оскільки така СКВ лише надає доступ до взаємодії з даними, то це дає можливість використовувати одну й ту ж СКВ для багатьох каналів доступу таких як веб-сайти, сторонні сервіси, мобільні та десктопні застосунки, розумні годинники та інші
- Низька ресурсна затратність – це рішення потребує менше ресурсів для інсталяції та підтримки аніж класичні СКВ, їх легко запускати і дуже часто це роблять в хмарі не відповідних сервісах, як AWS Cloud, Google Cloud, Digital Ocean та інші.
- Швидший час розробки продукту – такі СКВ відрізняються тим, що серверна частина відокремлена він клієнтської, отже і розробка може

відбуватися паралельно, тому проект зазвичай виконується в два рази швидше

- Легкі для використання – традиційні СКВ зазвичай досить складні для розуміння та комплексні, бо розробники намагаються вкласти якомога більше можливостей в своє рішення з коробки, а такий підхід лише фокусується на управлінні контентом, тобто весь принцип роботи спрощується лише до роботи з контентом
- Гнучкість – Веб-сервер і клієнт ніяк не пов’язані один з одним крім REST інтерфейсу яким вони спілкуються, тому як клієнт так і сервер можуть бути написані на будь-якій мові програмування з використанням будь-яких технологій і вироджується в можливість написання систем, що використовують мікросервісну архітектуру.
- Потужність – такі системи легко маштабуються, адже вони засновані на ідеях stateless API, тобто такого, що не зберігає проміжний стан на сервері і не потребує його наявності.
- Персоналізованість – рішення написані з використанням цього підходу є більш функціональними та кастомними, адже клієнтська частина розроблюється власноруч,А не генерується СКВ.

Серед недоліків варто навести:

- Багатосервісність – управляти декількома системами може бути досить складно і потребує більших технічних навичок від спеціалістів, тому це очевидно збільшує затрати на розробку такого рішення.
- Відсутність шаблонних компонент – оскільки такий підхід означає, що СКВ ніяк не приймає участь в створенні рівня відображення, то розробникам доведеться створити це все власноруч включаючи шаблоні речі, такі як навігацію веб-сайтом, пошук по сторінці, авторизація, реєстрації та інше.

- Організація контенту – також як результат відсутності СМА та концепту сторінок, то людям відповідальним за управління контентом потрібно адаптовуватись до факту, що контент існує в своїй чистій формі окремій від відображення і їх потрібно знати, де буде з’являтися те чи інше поле на сторінці.

1.3 Огляд наявних аналогів

Strapi (рис. 1.1) – безкоштовна Headless CMS з відкритими програмним кодом написана на JavaScript та Node.js з використанням фреймворків Koa для сервера та React для написання адмін панелі. Серед клієнтів, що використовують цю СКВ присутні: NASA, IBM, Walmart, Shelt.in та інші. Ця СКВ досить швидко набула популярності і стала лідером ринку [1].

Щоб почати працювати з Strapi достатньо створити проект з шаблону запустивши команду наведену на офіційному сайті та запустити додаток. Наступним кроком потрібно перейти на /admin сторінку та створити головного користувача системи, після чого можно приступити до створення моделі, що потрібна користувачу. Коли схема моделі буде готова достатньо зберегти зміни і сервер сам перезапуститься і далі вже можна буде перейти безпосередньо до роботи з контентом і редагувати сутності в редакторі з елементами вводу і виводу згенерованими згідно до схеми.

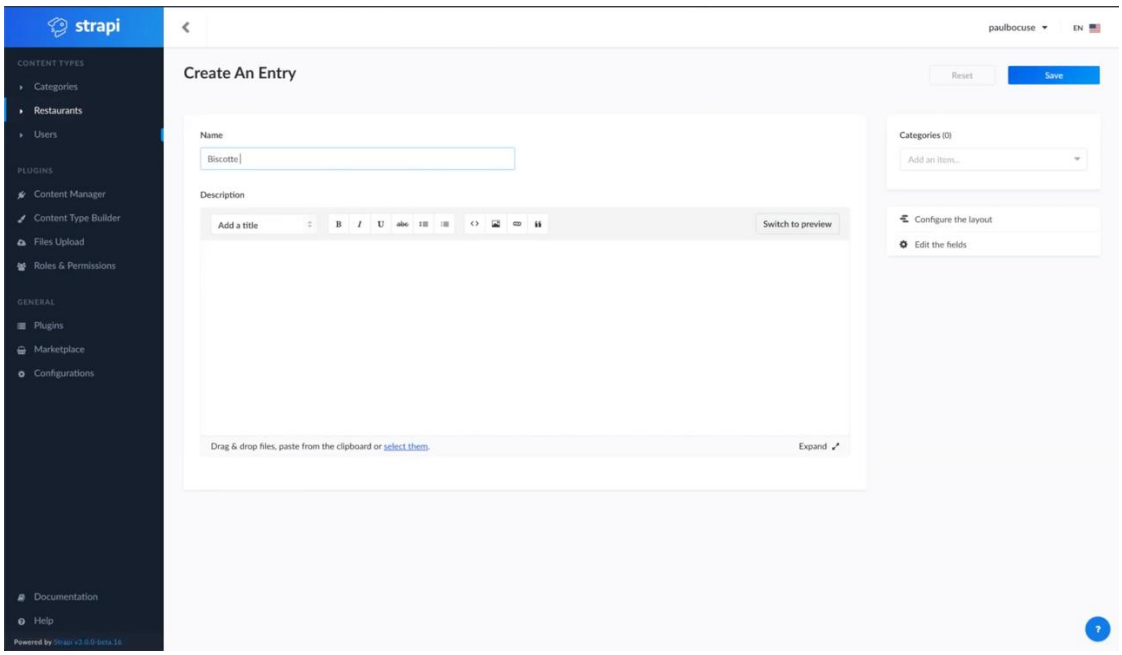


Рис. 1.1 Інтерфейс сервісу Strapi

Наразі ця СКВ є лідером ринку і воно і не дивно, бо це рішення не тільки безкоштовне, проте дуже вміло спроектоване і тому це сприяє продуктивній роботі з цією СКВ.

Основними недоліками є:

- Відсутність моніторингу стану сервісу
- Обмежений вибір способу зберігання даних
- Адмін панель працює тільки з одним сервісом і не передбачено можливості, щоб вона взаємодіяла з сервісами написаними на інших мовах програмування
- Відсутність повного контролю над тим, що відбувається при CRUD операціях
- Немає вбудованого перегляду документації прямо з СКВ
- Відсутність можливості експорту даних
- Немає функціоналу для виконання певних дій над вибраною кількістю моделей одночасно, тобто змінити можна тільки одну сутність за раз і не передбачено надання шаблонних дій

Перевагами є:

- Це безкоштовне рішення з відкритим кодом
- Редагування схеми прямо прямо в адмін панелі
- Можливість настроїти як API так і адміністративну панель
- Є самостійним рішенням, що користувач сам запускає та підтримує в ізоляції від інших користувачів СКВ
- Дає можливість взаємодіяти за допомогою REST або GraphQL, а також автоматично генерує документацію для них
- Розширюваність СКВ дає можливість встановлювати додаткові плагіни, що розширюють функціонал СКВ
- Захищеність: CORS, CSP, P3P, XSS, Xframe, Авторизація/Автентифікація за допомогою JWT токена
- Широкий спектр наявних типів полів включаючи можливість реляцій між сутностями
- Зручний, красивий та функціональний інтерфейс адмін панелі
- Висока швидкість системи
- Потужний інтерфейс для командного рядка, що дозволяє легко керувати додатками з Strapi
- Гнучкі можливості пошуку та фільтрації даних, їх пагінація та конфігурація відображення полів

Keystone (рис. 1.2) – друга за популярністю безкоштовна Headless CMS з відкритими програмним кодом, що також написана на JavaScript та Node.js, де адмін панель також написана з використанням React, а сервер написаний на Express. Має багато схожих особливостей як і у Strapi, проте деякі властивості відрізняються адже тут наявний дещо інший підхід до управління схемою даних [2].

Для створення проекту з використанням Keystone достатньо запустити в командному рядку одну npx команду, що створить проект з шаблону так само як і в випадку з Strapi, далі потрібно налаштувати користувача для входу і це є

зовсім не продуманим і потрібно робити це власноруч, тобто потрібно настроїти свій спосіб автентифікації і лише потім можна буде ввійти. На відміну від Strapi тут немає редактора схеми і щоб її створити потрібно зробити це в програмному коді, після опису схеми з'являється можливість керувати створеною сутністю.

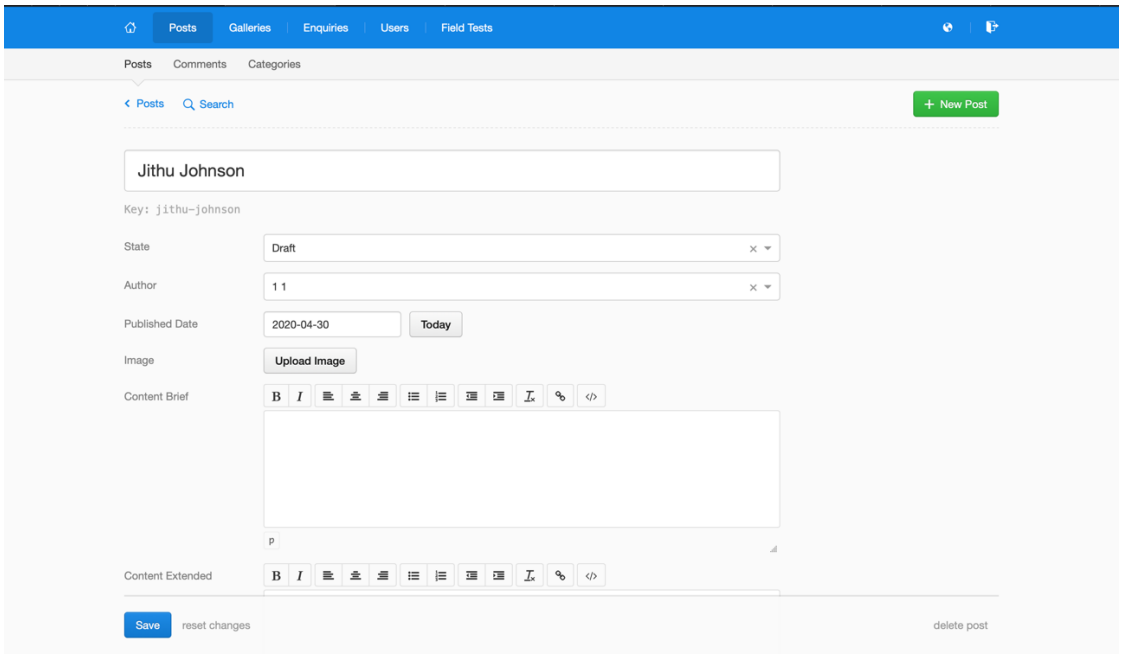


Рис. 1.2 Інтерфейс сервісу Keystone

Keystone можна сміливо назвати досить популярною та зручною СКВ з цілою купою можливостей, проте вона має спільні з Strapi недоліки.

Основними недоліками є:

- Як сховище для даних можна використовувати лише Postgresql та MongoDB на вибір
- Прив'язка до конкретної платформи і рішень, так званий vendor lock
- Відсутність повноцінних інструментів та способів розширювати СКВ, тобто автори не передбачили щось наподобі плагінної архітектури і користувач отримує все і одразу, проте без додаткового функціоналу, як це існує в Strapi
- Не відображається документація інтерфейсів СКВ з адміністративної панелі
- Відсутність імпортування даних в СКВ в JSON чи CSV форматах

Перевагами є:

- Keystone повністю безкоштовне рішення з відкритим кодом, тому будь-який недолік чи баг швидко усувається за допомогою потужного ком'юніті СКВ, також це означає, що нові можливості швидко додаються
- Є самостійним рішенням, а не знаходиться десь в клауді, тому клієнт має можливість власноруч розпоряджатися середовищем для інсталяції СКВ
- Швидкий пошук, редагування полів, що потрібно відображати, фільтри пошуку, наявність експорту в JSON та CSV форматах
- Підтримує широкий спектр різноманітних видів полів, реляцій, фотографій та інших файлів, також є поля формату WYSIWYG. Можна додавати свої власні типи полів.
- Настроюванні хуки, що запускаються після трьох основних дій створення, видалення та оновлення. Для кожного з цих етапів можна встановити хук при декодування вводу, валідації даних, перед та після виконанням дій
- Використання GraphQL, що вирішує проблеми over-fetching і under-fetching, тобто клієнт сам каже, які поля він хоче бачити, тому мережею передається менший трафік. Також це вирішує проблеми з документацією адже наявне таке поняття як інтроспекція, що надає повну інформацію про інтерфейс системи з якою взаємодіє клієнт, проте автори не спромоглися надати можливість перегляду документації прямо з СКВ
- Гарна документація СКВ та наявність онлайн демо, що сприяє стрімкому освоєнню системи і дає можливість швидко почати використовувати її повноцінно.

Contentful (рис. 1.3) – дуже популярна Headless СКВ з купою можливостей з коробки, проте це рішення кардинально відрізняється від попередніх, бо в цієї СКВ закритий програмний код і вся робота з нею відбувається через інтерфейс СКВ, що є частиною інфраструктури власників СКВ. Це означає, що дані зберігаються на їх серверах і доступ до них можна отримати або за допомогою наявного REST API або використовуючи вже написані розробниками СКВ готові SDK клієнти для роботи з нею. Іншим неприємним моментом є те що безкоштовно можна отримати лише дуже обмежений функціонал: 1 приклад додатку, 1 робочий простір, 10 користувачів СКВ і 8 SDK. По суті дають настільки мало, що аби будувати щось серйозне з використанням цієї СКВ потрібно купувати платну підписку, а це мінімум 37 доларів, щоб отримати 1 роль, 2 локалізації, 24 типи контенту і можливість зберегти 5 тисяч записів, ще є план за 879 доларів де надають 4 ролі, 10 локалізацій, 48 типи контенту і 50 тисяч записів і якщо і цього недостатньо, то існує варіант домовитись напряму щодо гнучкого плану підписки [3].

Почати користуватись Contentful досить просто: нажимаємо прямо на сайті сервісу пункт «Get started» і після набору стандартних питань і реєстрації через google, github чи напряму користувач потрапляє в адміністративну панель, де в меню “Content model” можна настроїти модель контенту з яким користувач хоче працювати. Наявні такі типи полів:

- WYSIWYG
- Текст
- Числа
- Дата та час
- Локація на карті
- Медіа файли
- Boolean
- JSON об’єкт

- Посилання на іншу модель контенту

Після того як модель створена можна почати наповняти її контентом. Для цього потрібно перейти в меню Content, де можна робити всі CRUD дії, наявна можливість к, хто наразі переглядає контент, а також можна продивитись метаінформацію, що потрібна Contentful. Крім того окремо є можливість працювати з медіа файлами. Можна встановлювати різноманітні плагіни, що вже написали автори СКВ. Важливим меню є “Settings” де можна не тільки конфігурувати СКВ, а й створити користувачів, що можуть робити запити в СКВ, настроїти ключі і права доступу до СКВ.

Недоліками такого підходу є:

- Vendor lock в питаннях зберігання даних, швидкості, доступності СКВ
- Обмежена безкоштовна функціональність і дуже високі ціни на платну підписку
- Відсутня можливість писати свій додатковий функціонал та неможливість реагувати на дії в СКВ з власною логікою
- Немає можливості переглядати документацію інтерфейсів СКВ для моделей

До переваг можна ж віднести:

- Швидкий початок роботи з СКВ та зрозумілий інтерфейс
- Непотрібно додатково витрачати гроші на базу даних і хостинг серверу, а також не потрібно витрачати час на їх вибір
- Можливість перегляду історії змін контенту
- Можливість коментувати контент
- Управління ролями та правами
- Велика кількість SDK для роботи з СКВ
- Гарний інтерфейс для пошуку такий же як у Strapi, Keystone
- Обширний набір доступних полів, проте не можна додавати свої

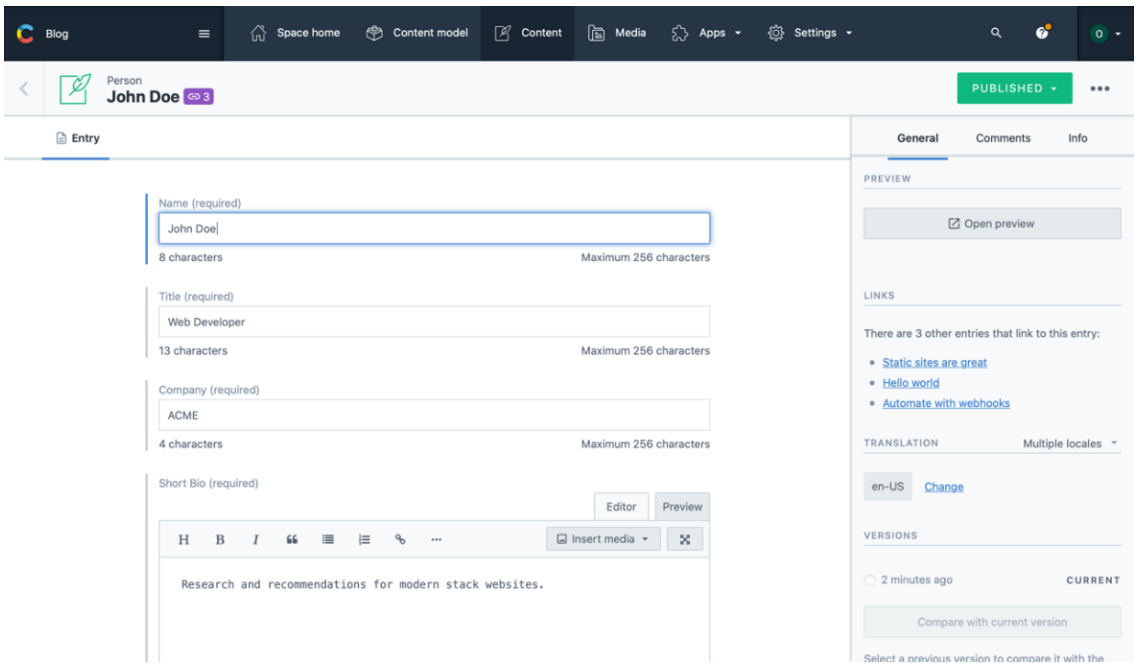


Рис. 1.3 Інтерфейс сервісу Contentful

WordPress (рис. 1.4) – це класична СКВ з відкритим кодом, саме вона стала тим рушієм, який привернув чималу увагу до СКВ і саме за допомогою цієї СКВ створено величезну кількість веб-сайтів різного формату починаючи від блогів і йдучи до більше складних варіантів. Вдала архітектура, вбудована система тем і плагінів, величезний простір можливих змін та розширень дозволяють конструювати на основі WordPress практично будь-які додатки. Написана мовою програмування PHP і суворо зав'язана на БД MySQL [4].

Щоб почати роботу з цією СКВ потрібно скачати WordPress з офіційного сайту, розмістити його на хостингі чи на власній машині, створити БД та користувача для доступу в БД. Далі проводиться базова конфігурація і запукається скрипт інсталяції, після чого можна починати працювати з СКВ. Додадатково можна встановити будь-які плагіни та теми, можна змінювати що завгодно в кодовій базі СКВ, бо постачається вона не як бібліотека, а як програмний код. Саме тому через таку особливу модель розповсюдження ця СКВ стала культовою і до сьогодні є дуже популярним і впливовим рішенням для будь-якої проблеми.

Переваги:

- Велика кількість безкоштовних плагінів та тем і сама СКВ безкоштовна
- Зручний, багатий на функціонал адміністративний інтерфейс, включаючи пошук, CRUD, кастомні дії
- Величезний простір для кастомізації, бо СКВ розповсюджується як програмний код
- Часті оновлення
- Висока продуктивність СКВ
- Управління користувачами, ролями та правами

Недоліки:

- Досить важко почати працювати з СКВ, це потребує чималого розуміння того як СКВ побудована, її структура програмного коду, архітектурні рішення, тощо
- Проблеми з захищеністю. Ці проблеми часто виникають, бо плагінів і тем дуже багато і досить легко потрапити в руку зловмисників, що розповсюджують теми і плагіни з якимись злоякісними скриптами
- Прив’язка до мови програмування. СКВ написана на PHP і тільки на ньому і можна працювати з нею.
- Сильна зв’язаність серверу від клієнтської частини. Адміністративна панель відображається за рахунок того ж PHP і це створює сильну пов’язаність
- Також немає можливості переглянути документацію інтерфейсів WordPress для цього потрібно добре знати її внутрішній устрій. Лише таким чином можна зрозуміти, які є доступні інтерфейси і як до них звертатись

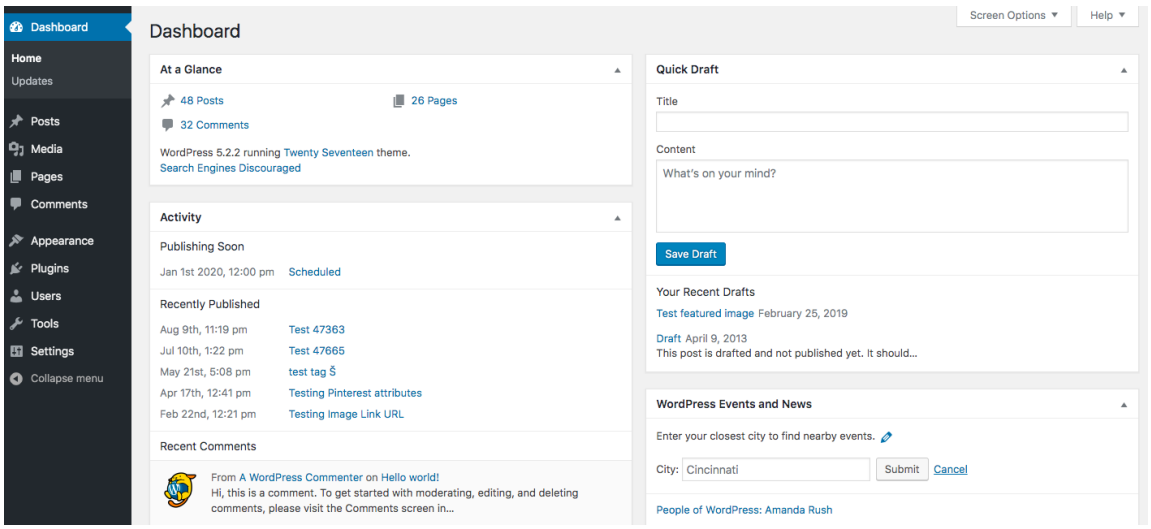


Рис. 1.4 Інтерфейс сервісу WordPress

ButterCMS (рис. 1.5) – це ще один приклад Headless CMS, що надається користувачам, як SaaS рішення, що включає одразу значний функціонал, дає можливість створювати свої моделі і вже завоювало чималу кількість прихильників. СКВ бере на себе всі клопоти створення інфраструктури для сервера на себе і надає дуже зручний та продуктивний інтерфейс для управління контентом. При цьому з старту є готове рішення для створення блогу, можна створювати шаблони даних для сторінок, щоб згодом власноруч відобразити контент такої сторінки [5].

Принцип роботи з цією СКВ дуже схожий на принцип роботи з Contentful, тобто після реєстрації користувач обирає план роботи з СКВ – безкоштовний чи платний і він потрапляє в адміністративну панель, де побачить свій ключ доступу до REST API інтерфейсу за допомогою якого можна робити запити до СКВ. Також доступні готові SDK для роботи з СКВ для більш ніж 20 мов програмування. Далі користувач взмозі управляти контентом блогу, сторінок або створити свою власну модель, яку він хотів би наповнювати контентом в своїй СКВ.

Серед типів полів наявні:

- Короткий та довгий текст
- Число

- WYSIWYG
- Посилання на модель
- Дата та час
- Випадаючий список з обмеженими елементами
- Медіафайл
- Умовний тип Boolean
- Html

З першого погляду рішення може здатись досить влучним, проте в нього є ряд недоліків:

- Vendor lock знову автори СКВ намагаються нав’язувати користувачам свої рішення і змушують сильно залежати від них
- Слабка розширюваність системи, не передбачено розширення ні силами авторів СКВ, ні силами користувачів
- Невеликий вибір типів полів для моделей
- Відсутність пошуку, фільтрації, сортування, пагінації
- Обмежений функціонал, якщо використовується безкоштовна підписка при цьому можливість створювати контент через API можна отримати лише зв’язавшись напряму з авторами СКВ і це коштуватиме більше ніж 208 доларів в місяць

До плюсів варто віднести такі властивості:

- Дуже ефективний та гарний інтерфейс СКВ, що сприяє швидкому старту
- Гарна документація авторів щодо користування СКВ
- Користувачу не доведеться ламати голову над вибором хостингу та сховища для даних
- Наявні готові рішення для створення блогу, веб-сторінок, а також значною мірою розкриті ідеї використання Headless CMS для побудови своїх додатків

- Управління користувачами та їх правами(Проте ролей не дуже багато, що можна було б вказати в якості мінуса)
- Обширний список готових для використання SDK для роботи з СКВ, а також зрозумілий REST API на випадок, якщо немає SDK або не сподобався його інтерфейс

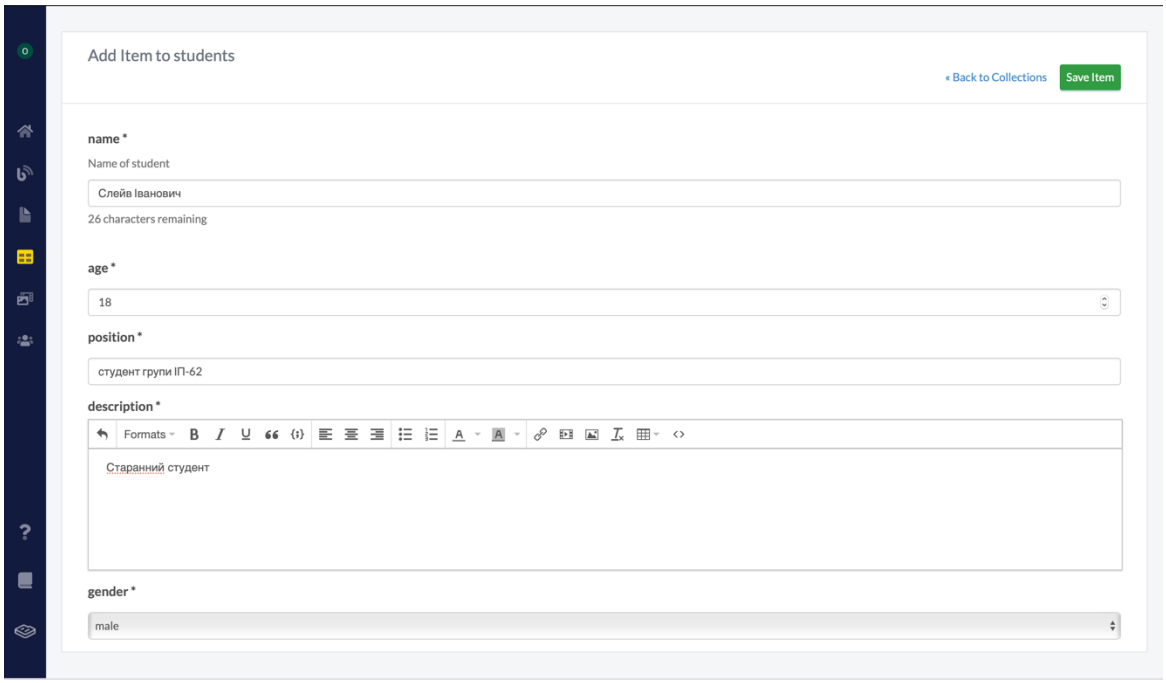


Рис. 1.5 Інтерфейс сервісу ButterCMS

ВИСНОВКИ ДО РОЗДІЛУ 1

Проведений огляд наявних сучасних систем управління контентом, показав, що:

1. Основним завданням СКВ є зробити комфортним і зручним роботу з контентом: додавання, зміну, видалення, пошук, експорт та імпорт.
2. Поява великої кількості однотипних рішень для роботи з специфічним призвели до появи великої кількості різноманітних СКВ, що не схожі одна на одну і кожна по своєму особлива
3. Проаналізувавши готові реалізації СКВ стали зрозумілими такі недоліки:
    - Велика кількість СКВ є платними або дають певний функціонал лише за додаткову плату;
    - Відсутній функціонал слідкування за здоров'ям сервіса
    - Немає можливості переглядати документацію інтерфейсів серверу.
    - Сильна прив'язаність до можливих варіантів зберігання контенту.
    - Відсутність функціоналу імпорту, експорта та виконання дій над однією чи декількома елементами.
    - Слабка розширюваність та конфігурація цих СКВ
4. Дивлячись на вище згадані недоліки, можна визначити основні функції системи управління контентом:
    - Має реалізувати Headless CMS підхід, тобто клієнтська частина повинна бути незалежна від серверної
    - Зручний, приємний та зрозумілий API та UI інтерфейси
    - Повністю безкоштовна СКВ з відкритим кодом та повним доступом до всіх функцій;
    - Можливість виконання як CRUD операцій, так і створення своїх власних користувацьких дій, імпорт, експорт
    - Перегляд стану здоров’я та документації сервіса

- Зручний пошук та можливість обирати, які поля демонструвати
- Авторизація та автентифікація
- Управління правами доступу
- Версіонування контенту
- Можливістю роботи адміністративної панелі СКВ з багатьма серверами СКВ
- Робота з контентом має відбуватись після опису схеми моделі на серверній частині, після чого клієнтська буде її отримувати і буде на льоту будувати адміністративний інтерфейс для роботи з моделлю відповідно до схеми.
- СКВ має вміти працювати з такими типами полів: умовний, текстовий, число, перелік, JSON, список, дата та час, WYSIWYG, реляції, локації

РОЗДІЛ 2
ПРОЕКТУВАННЯ СИСТЕМИ УПРАВЛІННЯ КОНТЕНТОМ

2.1 Структура системи управління контентом

Розроблювана система управління контентом представляє з себе клієнт-серверний додаток, де сервером є веб-сервер СКВ, а клієнтом є або користувач, що відкриває адміністративну панель в веб-браузері, або користувач, що управляє контентом напряму через інтерфейси СКВ.

Клієнтська частина відповідає за взаємодію з користувачем, візуалізацію даних, проводить базову валідацію введених даних, спираючись на права користувача вона також здатна приховувати інформацію про функціонал, що недоступний користувачу. Адміністративній частині доводиться виконувати досить тяжку та абстрактну роботу по роботі не з чітко визначеними при створенні клієнтської частини моделями, а з такими про які вона дізнається тільки після взаємодії з серверною частиною, до того ж це може бути не один сервер СКВ, декілька і ще й написані на різних мовах програмування. Саме для цього було розроблено спеціальний протокол спілкування серверу з клієнтом: клієнт запитує у сервера доступні моделі і сервер віддає схеми, що описують ці моделі саме завдяки цьому на основі цих схем, що включають інформацію про всі поля та можливі дії, клієнтська частина здатна відображати правильні форми для управління контентом та виконанням інших дій. Клієнт взаємодіє з сервером завжди передаючи спеціальний секретний спеціальний для даного користувача ключ, що сервер перевіряє на чинність і лише тоді відповідає клієнту.

Серверна ж частина відповідає за обробку запитів надісланих клієнтською частиною, вона відповідає про оголошення і розповсюдження інформації про схему контенту і доступних моделей, надає доступ до своїх інтерфейсів виконання CRUD операцій на моделями, інтерфейсу дій,

розповсюджує інформацію про здоров’я та документацію серверу. Також на серверну частину потрапляє обов’язок зберігання даних і цей момент є повністю гнучким і користувач має можливість обрати будь-який варіант зберігання даних: будь-яка БД, якийсь кеш, просто в оперативній пам’яті чи будь-який інший спосіб, яким бажає скористуватись користувач.

Взаємодіють клієнт та сервер шляхом спілкування за допомогою протокола HTTP і роблять вони це слідуючи REST підходу. Немає додаткових внутрішніх прошарків всі дані пересилаються в тому вигляді, в якому їх буде зручно використовувати клієнту, в даному випадку це JSON. Доступ до кожного ресурсу чи дії можна отримати за певним посиланням URL при цьому для різних дій використовуються різні статус коди, що існують в HTTP протоколі, а саме: Get для отримання даних, Post – для додавання даних, Put – для зміни існуючих даних та Delete – для видалення даних.

Використовується Stateless підхід і сервер не зберігає жодної інформації про клієнта і вся необхідна для виконання дій інформація знаходиться в запиті до серверу СКВ.

Система управління контентом має дуже гнучку структуру і по суті складається з модулів Admin UI, Admin API та DB. Admin UI та Admin API структурно складаються з іншим модулів, детальний опис, яких знаходиться в Додатку 1.

Докладніше про кожен модуль системи:

- **Admin UI** – це модуль адміністративної панелі СКВ, представляє собою клієнтську частину застосунку. Цей модуль надає функціонал відображення інформації та обробки дій користувача, а саме: відображення документації, здоров’я серверу, механізми управління та перегляду контенту, управління ролями, контролем версій. Модуль оброблює запити користувача та перетворює їх в відповідні HTTP запити до модуля Admin API. Ця взаємодія відбувається після того, як

модуль Admin API надасть всю необхідно згідно з протоколом інформацію клієнту.

- **Admin API** – це модуль, що є частиною HTTP сервера СКВ, вміє конфігурувати СКВ, зберігає документацію, опитує компоненти про їх стан для надання інформації про здоров'я сервера, реалізує протокол спілкування СКВ і має всі потрібні інтерфейси для взаємодії з Admin UI. На запит останнього Admin API віддає інформацію про моделі наявні на сервері і далі оброблює його запити. Крім опису контенту та інших деталей модуль взаємодіє з модулем DB шляхом реалізації на стороні сервера інтерфейсу спілкування з тією чи іншою DB. Для кожного типу контенту можна використовувати різні бази даних.
- **DB** – база даних, що виконує роль модуля для зберігання контенту. Зберігає дані у довільному вигляду, який визначається особливостями обраної бази даних. Якщо цей модуль присутній, то зазвичай виконує функції додавання, зміни, перегляду, видалення та пошуку даних, що зберігаються в базі даних.

Схема взаємодії модулів представлена на рис. 2.1

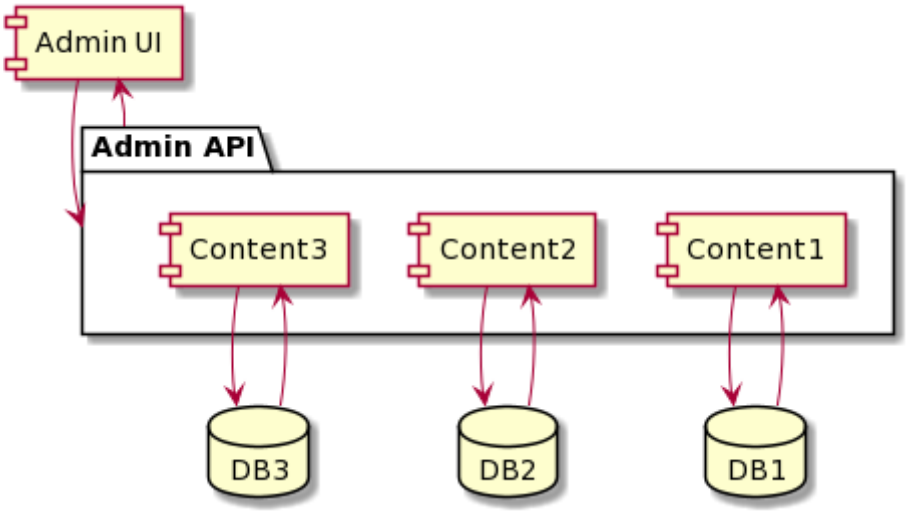


Рис. 2.1 Схема взаємодії модулів СКВ

2.2 Серверна частина системи

При огляді існуючих СКВ рішень було виявлено, що найкращим рішень при написанні Headless CMS є створення протоколу, що визначає схему і на її основі відбувається подальша робота з СКВ, також дуже важлими моментами є швидкодія СКВ, гарна спільнота з великою кількістю бібліотек і багатим синтаксисом, що зробить процес написання і конструювання елементів СКВ швидким та зручним процесом і дасть можливість розробнику обирати власноруч як зберігати дані та який HTTP сервер використовувати. При цьому крім цього перша серверна реалізація буде гарним приклад того як мають виглядати реалізації серверної частини СКВ для інших мов програмування. Тому при виборі мови програмування для створення серверної частини СКВ мені довелось враховувати цю інформацію і було обрано мову програмування Scala.

Scala – це мульти-парадигмова мова програмування, що поєднує риси об’єктно-орієнтованої мови та функціональної мови. Це мова програмування загального призначення, що дає можливість вирішувати широкий спектр задач завдяки своєму багатому синтаксису, функціоналу та неймовірно продуманому компілятору. Є можливість запускати програми як на віртуальній машині Java так і інтерпритувати як JavaScript або компілювати програмний код одразу в нативний машинний код. Мова пройшла досить довгий життєвий шлях і наразі знаходиться в процесі переходу до її третьої версії компілятор якої зветься Dotty. Навколо цієї мови програмування зібралось досить велика і дружня спільнота розробників і саме завдяки цьому існує велика кількість бібліотек і програмного забезпечення написаного мовою програмування Scala включаючи бібліотеки для роботи з будь-якими БД, для роботи з JSON та фреймворки для створення веб-додатків. До того ж оскільки Scala може працювати на JVM це дає можливість використовувати будь-які JVM бібліотеки при компіляції під JVM платформу. Іншим ключовим аспектом при виборі було наявність засобів

для метапрограмування, а саме макросів, що дають змогу оперувати AST деревом мови на етапі компіляції, що відкриває простір до використання цього при написанні схеми для моделей. Одна з найпривабливіших особливостей цієї мови це любов спільноти до функціонального програмування, використання теорії категорій на практиці, практичне застосування монад, семігруп, функторів, аплікатив та інших понять, що Scala спільнота запозичила з теорії категорії. Ці поняття і наявність інструментів для реалізації зручного ad-hoc поліморфізму, що базується на тайпкласах і чимала їх кількість вже реалізована в двох популярних бібліотеках Scalaz та Cats. В рамках цього проекту було обрано бібліотеку Cats, бо наразі вона є лідер ринку і найбільша кількість бібліотек використовують саме її [6].

Коли мова йде про серйозний проект, то дуже тяжко уявити його без гарної продуктивності та швидкодії, а цього досить тяжко досягти пишучи синхронний код, саме потрібні інструменти для написання асинхронних та конкурентих програм і Scala пішла досить далеко в розвитці цих понять. В стандартній бібліотеці Scala не тільки доступно все те, що існує для вирішення цих задач в Java, але й наявний свій тип під назвою Future, його особливістю є те, що він є монадичним і на ньому визначені операції map, flatMap та купа інших зручних методів, що роблять написання таких програм дуже зручним та зрозумілим [16]. До того ж в Scala існує for comprehension який дає змогу перетворити набір викликів map та flatMap в набір однострічкових викликів, що перетворює програму з вкладених один одну функцій в щось, що виглядає як імперативний код. Наприклад код асинхронного додавання двох чисел на рис 2.2 можна перетворити в еквівалентний вираз як на рис 2.3, проте більше виразний.

```
future1.flatMap(value1 => future2.map(value2 => value1 + value2))
```

Рис. 2.2 Асинхронне додавання чисел

```
for {
  value1 <- future1
  value2 <- future2
} yield value1 + value2
```

Рис. 2.3 Асинхронне додавання чисел з for comprehension

Scala завжди надавала перевагу абстракції ніж конкретиці і це питання не виключення. Для того, щоб абстрагуватися від типу ефекту(так надалі будуть називатись всі асинхронні чи конкурентні числення) було придумано стиль під назвою Tagless Final. Суть цього підходу заключається в тому, що методи і класи, які виконують якісь асинхронні чи конкурентні дії повинні оголошувати в своїй сигнатурі generic тип під назвою F при цьому він має бути екзистенціальним типом. Прикладом оголошення методу в такому підході є рис. 2.4

```
def add[F[_]](effect1: F[Int], effect2: F[Int]): F[Int] = ???
```

Рис. 2.4 Сигнатура метода додавання в стилі Tagless Final

Простими словами оголошується тип F, який здатний зберігати в собі будь-який інший тип, таким чином сюди можна підставити будь-який тип, що має одну дірку будь-то Future, Option чи будь-який інший тип. Проте просто таке оголошення не дає можливості виконувати якісь операції над екземплярами такого типу, саме для цього вирішили скористатись уже існуючими тайпкласами, а також дописати нові, які б описували асинхронні та конкуренті дії саме так і виникла бібліотека Cats Effect, яка використовується в написані СКВ для абстрактного опису ефектів, що можуть виконуватись в системі. Завдяки наявності в Scala функціоналу під назвою implicits існує можливість неявної передачі параметрів в функції, якщо існує оголошення

неявної змінної чи метода і саме цю властивість використовують при написанні Tagless Final програм. Щоб отримати можливість виконувати дії над абстрактними типами методи очікують неявної передачі тайпкласу який дає можливість виконувати ті чи інші дії. Спробуємо закінчити метод додавання абстрагований від типу ефекту (рис. 2.5)

```
def add[F[_]: Monad](effect1: F[Int], effect1: F[Int]): F[Int] =
    for {
      value1 <- effect1
      value2 <- effect1
    } yield value1 + value2
```

Рис. 2.5 Функція додавання чисел абстрагованих від обгортки в стилі Tagless Final

На подив програма майже не змінилась ми лише просимо неявний параметр тайпкласу Monad і завдяки implicit класам для типів для яких наявний неявний параметр Monad існує синтаксис виконання методів map та flatMap, що дає змогу використовувати їх в for comprehension. Тобто на виході ми отримуємо абстраговану від типу ефекту програму суть і вигляд якої майже не змінився і можна викликати її з будь-яким типом ефекту для якого оголошений неявний параметр тайпкласу Monad [7].

При написанні бібліотеки чи рішення, яким будуть користуватись інші розробники, а в нашому випадці це СКВ, використання цього підходу є необхідністю, бо в Scala екосистемі найчастіше використовується не бібліотечний тип Future, а інші готові типи ефектів з відкритим кодом, такі як: ZIO, Cats-Effect IO, Monix [8][9][10]. Оскільки кожен автор чогось, що будуть використовувати інші не хоче нав’язувати своє бачення того, яку реалізацію ефектів потрібно використовувати, то саме Tagless Final підхід вирішує цю задачу і єдине, що потрібно, щоб почати користуватись написаними

абстрактними методами – написати реалізацію тайпкласів, які неявно потребують такі методи

Будь-який проект на скала потребує спеціального збірника, що виконує роль збирання, конфігурування проєкта, управлінням залежностей та виконує інші дії пов’язані з інфраструктурою. Спочатку в Scala його не було і доводилось використовувати існуючі рішення: Gradle, Maven. Проте це швидко змінилось і наразі найкращим збірником є Scala Build Tool, коротко sbt. Після встановлення sbt проект можна створити за допомогою генерації проекту з шаблону або згенерувавши новий пустий проект. Всі файлики з розширенням «.sbt» будуть автоматично виконанні збірником, якщо вони знаходяться в корені проекту або в теці project. Коренем проекту вважається тека в якій знаходиться файл з назвою build.sbt зазвичай саме в ньому знаходиться вся конфігурація проекту [14].

Оскільки для створення СКВ потрібно працювати з HTTP потрібно обрати відповідний фреймворк для побудови HTTP сервера і наразі існує 3 основні такі рішення, а саме: Http4s, Akka Http, Finatra. Тут є одна дилема, оскільки наявне бажання дати можливість користувачу самому обирати сервер на якому він хоче запускати СКВ, то потрібно мати можливість мати проміжне представлення наших HTTP інтерфейсів, яке потім можна буде перетворити в щось з чим може працювати той чи інший HTTP фреймворк. Вирішення проблем такого роду є дуже популярною задачею в Scala спільноті і конкретно для цієї задачі вже є бібліотека під назвою Tapir [11]. Саме її було обрано для декларації інтерфейсів СКВ. Tapir дає можливість описувати HTTP інтерфейси як іммутабельні Scala об’єкти шляхом використання гнучкого DSL, який дає змогу описувати інтерфейси як набір вхідних параметрів, вихідний тип відповіді та набір помилок, якими здатен відповідати інтерфейс. Цей опис можна перетворити в формат OpenApi і використати його для демонстрації документації інтерфейсів, що є однією з задач, що має виконувати СКВ. Після того як опис інтерфейсу готовий користувач бібліотеки здатен створити

специфічну структуру з якою вміє працювати Http4s, Akka Http для цього викликавши функцію відповідну для HTTP бібліотеки потрібно надати функцію, яка виконує так звану бізнес логіку, кількість та типи її аргументів мають відповідати типам і кількості вхідних аргументів опису інтерфейсу, а на вихід вона має віддавати асинхронний результат, що є або успішною відповіддю або помилкою. Сигнатура метода виглядає так: I => F[Either[E, A]], де I – вхідні параметри, F[_] – це тип ефекту, Either – це тип, що описує щось здатне бути або лівим типом або правим, тобто це або E – помилка або A – успішний результат. Таким чином всі інтерфейси СКВ будуть описані в Tagless Final стилі з використанням Tapir і саме в момент запуску застосунку розробник буде обирати реалізацію HTTP серверу та тип ефекту з яким він бажає працювати.

Незважаючи на те, що бібліотека Tapir абстрагує нас від вибору HTTP серверу все таки важливо знати, що з себе представляють наявні реалізації.

Http4s – це повністю функціональна реалізація HTTP сервера та клієнта, він включає в себе модель HTTP сутностей запити і відповіді, написаний повністю в функціональному стилі з використанням підходу Tagless Final базованого на використання бібліотеки cats-effect, підтримує стрімінг даних, middleware та включає велику кількість різноманітних модулів, що надають ті чи інші функції. Базується ця бібліотека на власній реалізації HTTP серверу під назвою Blaze [12].

Akka Http – це набагато старша реалізація Http сервера і вона розрахована лише на тип Future і тут не передбачено абстракцію над типом ефекту, проте це дуже добре спроектована бібліотека, що включає свою реалізацію Http моделі, великий набір підходів до роботи з запитами і можливістю відповідати широким спектром типів контенту. Бібліотека також має чималий набір додаткових модулів, що надає додатковий функціонал. Обидві бібліотеки мають майже ідентичний функціонал за виключенням того, що http4s абстрагований від типу ефекту і з коробки дає набір готових рішень для

стандартних задач HTTP серверів, таких як кешування, CORS, роздача статичних файлів і т.д [13].

Для роботи з JSON було обрано бібліотеку Circe, вона є повністю функціональною реалізацією роботи з цим форматом і надає свою реалізацію AST дерева, що представляє собой об’єкти та класи, що є аналогами JSON типів. Також є модулі серіалазіції та десеріалізації Scala типів в JSON, що представлені тайпкласами Decoder та Encoder та сумою цих типів під назвою Codec. Реалізовувати їх можна як вручну за допомогою низькорівневого DSL шляхом розбору курсору, що рузається по структурі JSON об’єкта, так і автоматично за допомогою готових макросів, що ще на етапі компіляції створюють все потрібне.

Оскільки для створення схеми було обрано скористатись метапрограмування, а писати низькорівневий макрос немає потребо, було прийнято рішення скористатись бібліотекою під назвою Magnolia. Її суть полягає в тому, що вона спрощує автоматичну генерацію тайпкласів для класів та sealed ієрархій. Нас цікавить лише випадок з класами і для генерації схеми ця бібліотеки підходить ідеально, бо для кожного поля класу вона здатна знайти неявний параметр типу FieldSchema і тип самим ми можемо автоматично ще на етапі компіляції отримати автоматично згенеровану схему.

Очевидним є момент, що користувач СКВ захоче скористатись SQL базою даних незважаючи на те, що СКВ дає повну свободу вибору сховища даних завдяки своєму дизайну, гарною ідеєю буде реалізувати декілька готових варіантів зберігання контенту. Було прийнято рішення написати реалізацію для будь-якої SQL БД і для цього ми скористаємось бібліотекою Slick. Після опису таблиці сутності за допомогою інструментів бібліотеки ми повині вказани, як поля таблиці пов’язані з класом, що представляє нашу модель і далі шляхом виклику потрібних методів на об’єкті типу TableQuery[Table] ми можемо описувати взаємодію з БД, що має тип DBIO[Result], а згодом запустити цей опис на виконання, скориставшись методом типу Database під назвою run.

Бібліотека абстрагована від особливостей БД з якою вона працює, тому можна спокійно написати модуль для роботи з нашою СКВ і розробник згодом сам вкаже Profile БД з якою він працює [15].

Для аутентифікації застосунку використовується JSON Web Token, коротко JWT. Для Scala існують готові рішення для роботи з цим форматом і була обрана реалізація з відкритим програмним кодом від автора pauldijou. Це найпопулярніша бібліотека для роботи з цим форматом, а також вона має підтримку вже обраної бібліотеки для роботи з JSON - circe. Суть JWT заключається в тому, що інформація про токен вшивається в JSON об'єкт та шифрується обраним шифром з використанням секретного ключа. Токен складається з заголовка, вмісту та підпису. Перші два формуються окремо, а на їх основі формується підпис. Заголовок - JSON об'єкт, що має поля typ та alg, що вказують на тип токена та на алгоритм шифрування. В вмісті стандартно є такі поля:

- iss — (issuer) видавець токена
- sub — (subject) "тема", призначення токена
- aud — (audience) аудиторія, отримувачі токена
- exp — (expire time) термін дії токена
- nbf — (not before) термін, до якого токен не дійсний
- iat — (issued at) час створення токена
- jti — (JWT id) ідентифікатор токена

Інші поля додаються по бажанню у вільному вигляді, саме так в рамках даної роботи після авторизації в токен вшивається вся інформація про користувача в полі «user».

Для форматування коду використовується бібліотека ScalaFmt, яка встановлюється в конфігурацію sbt. Запускати процес форматування можна за допомогою команди fmt. Настройки форматування відповідають сучасним стандартам комерційної розробки і їз можна кастомізуват в файлі .scalafmt.conf.

2.3 Клієнтська частина системи

При написані адміністративної панелі було вирішено, що клієнтська частина має представляти собою SPA, тобто односторінковий додаток, який взаємодіє з користувачем динамічно змінюючи інтерфейс поточної сторінки, а не завантажує нову. Цей підхід наразі є найпопулярнішим рішенням при написані прогресивних застосунків, яким характерно мати зручний, динамічний та багатий на функціонал інтерфейс. Веб-додатки такого формату надають кращий користувацький досвід і зглажують приховані недоліки будь-якого клієнт-серверного додатку. Плавні переходи і спінери завантаження можуть вправно відвести увагу користувача від розуміння довгої відповіді від серверу. Щоб писати такі веб-додатки зазвичай використовують JavaScript, який здатний маніпулювати DOM сторінки динамічно, проте в наш час чистий JavaScript використовують лише ентузіасти і люди, які досі не знають, що існують готові фреймворки, що спрощують процес написання SPA додатків. Гарними прикладами таких фреймворків є Angular, React, Vue, Ember та інші. Кожен з фреймворків по своєму вдалий, проте серед них дуже виділяється React і саме йог обуло обрано для написання клієнтської частини СКВ.

React – це фреймворк для розробки прогресивних клієнтських веб-додатків розроблений компанією Facebook. Він включає в себе всі необхідні інструменти для побудови сучасного веб-додатка і дає на вибір два підходи побудови застосунку компонетно-орієнтований і більше функціональний варіант під назвою Hook API. В залежності від підходу відрізняється і підхід до розробки застосунку, Hook API є більше молодим рішенням, проте він вже встиг зарекомендувати себе, як заміна компонентного підходу. Для побудови СКВ було обрано саме Hook API [17].

Проект на React створюється з шаблону за допомогою пакетного менеджера npm і надалі в package.json будуть прописуватись всі залежності і конфігурація клієнтської частини проекту. Щоб створити компонент, який

можна використовувати при написанні інтерфейсу достатньо декларувати функцію ім’я якої починається з великої букви і вона має повертати шаблон, який буде відображатися при використанні цього компонента. Шаблон зазвичай описується за допомогою спеціального синтаксису, що зветься JSX, суть полягає в тому, що JSX опис не є ні текстом, ні HTML - це по суті проміжний елемент, що є звичайною декларацією шаблону, який згодом інтерпритує React своїм движком для управління DOM. Оскільки декларація компонентів представляє собою функцію, а JSX – це просто набір Javascript об’єктів, що можна присвоювати в зміні і робити з ними будь-що – це створює дуже гнучкий підхід до написання інтерфейсу. В JSX синтаксисі наявно поняття інтерполяції і в разі, якщо потрібен умовний оператор чи цикл, то їх можна вбудовувати прямо код шаблону шляхом інтерполяції.

Суть Hook API в тому, що користувача доступні такі методи, які називають хуки: useState, useContext, useEffect, useReducer, useCallback, useMemo, useRef та інші. Суть цих хуків заключається в тому, що кожен з них певним чином інкапсулює логіку управління змінними і подіями, що відбувають в системі і дає можливість дуже зручно писати додаток. Фундаментальними тут є перші три хуки.

useState приймає параметром початковий стан змінної і повертає кортеж геттера і сеттера для цього поля при цьому сеттер виконує операцію присвоєння асинхронно. Прийнято використовувати в JSX шаблони саме геттер useState хука, адже при зміні стану змінної шаблон знову оновиться.

useContext хук використовується при роботі з концептом контекстів, що полягає в тому що десь на високому рівні ініціалізується компонент в якості того хто надає контекст і будь-який вкладений компонент може отримати дані, які розповсюджує контекст за допомогою цього хука, наприклад в цій СКВ контексти будуть використовуватись для запам’ятовування користувача, http клієнта з ключем доступу користувача.

useEffect хук, що використовується для умовного виконання ефекту при оновленні сторінки, таким чином перший аргумент це ефект, який буде виконуватись при її оновленні, а другий це аргументи за умови зміни яких потрібно запускати наведений в першому параметрі ефект. Зазвичай використовується при ініціалізації компонента.

При розробці клієнтської частини веб-додатку було використано бібліотеку Material UI, що по собі являє собою реалізацію поняття Bootstrap розробленого компанією Facebook. Суть полягає в тому, що розробнику інтерфейса не потрібно витрачати багато часу на дизайн і стилізацію компонентів, вони одразу коробки дуже добре виглядають і наявно досить багато готових полів вводу і виводу, які знадобляться при створенні інтерфейсу роботи з моделями СКВ.

Ключовим моментом розробки адміністративної частини СКВ було виконання CRUD дій та пошуку і по суті інтерфейс мав би виглядати як таблиця з здатністю додавати записи, редагувати існуючі, видаляти їх, обирати декілька та виконувати дії і це було б досить великим куском роботи, проте знайшлось готове рішення під назвою material-table.

Також додатково для роботи з WYSIWYG було використано бібліотеку tinymce. Суть WYSIWYG заключається в тому, що одразу при редагуванні контенту користувач бачить контент в тому вигляді в якому він буде відображатись на сторінці у інших користувачів. В рамках цієї компоненти існує набір інструментів для роботи з текстом наподобі меню в Microsoft Word і користувач шляхом використання цих інструментів форматує контент і одразу бачить результати. В той же час компонента зберігає контент у вигляді HTML який і зберігається в базу даних. Це досить популярна річ для типографії, блогів.

Бібліотека json-editor надає можливість зручно редагувати JSON об'єкти, переглядати їх вміст, робити складні трансформації, пошуки, тощо. Бібліотека здатна робити майже будь-які операції над JSON і має дуже зручний інтерфейс. Оскільки СКВ підтримує цей формат поля було обрано саме цю бібліотеку.

Для роботи з полям з типом дати було обрано бібліотеку date-fns, що вміє працювати з стандартним JS типом для дат та надає зручний інтерфейс для роботи з датами.

ВИСНОВКИ ДО РОЗДІЛУ 2

В даному розділі було розглянуто рішення прийняті під час проектування системи управління контентом. Було наведено які сценарії має підтримувати дана СКВ і спираючись на специфіку задачі було обрано технологічний стек.

Веб-застосунок має клієнт-серверну архітектуру саме тому для цих двох частин були обрані певні інструменти розробки наведені далі.

Для клієнтської частини це:

- JavaScript – в якості мови для написання логіки
- Npm – пакетний менеджер проекта
- React – як основний фреймворк для створення елементів СКВ з використанням функціонального підходу під назвою Hook API
- Material UI – дає можливість отримати зручний та красивий інтерфейс без додаткових турбот про дизайн та стилізацію, а також купу готових рішень для елементів управління користувацького інтерфейсу

Для серверної частини це:

- Scala – мова програмування з багатим синтаксисом, можливістю метапрограмування, великою спільнотой і цималою кількістю існуючих бібліотек
- Sbt – система збирання проекту та управління його залежностями
- Tapir – бібліотека для опису API інтерфейсів застосунку з можливістю отримання документації для них та можливістю обирати на якій реалізації HTTP сервера будуть працювати ці інтерфейси. Серед можливих варіантів Http4s, Akka Http та Finatra
- Circe – робота з JSON форматом
- Magnolia – бібліотека, що є інструментом для метапрограмування в Scala і буде використана для генерації схеми моделі

- Slick – бібліотека для роботи з SQL БД абстраговано від імплементації
- Akka Http та Http4s – фреймворки реалізації HTTP серверу

РОЗДІЛ 3
РЕАЛІЗАЦІЯ СИСТЕМИ УПРАВЛІННЯ КОНТЕНТОМ

3.1 Аналіз протоколу спілкування СКВ

Дана СКВ є розподіленою реалізацією СКВ, що не обмежена одним сервером і однією мовою для якої вона реалізована ці унікальні можливості досягаються шляхом використанням особливого протоколу спілкування сервера з клієнтом.

Клієнтська частина звертається до серверної і запитує інформацію про доступні моделі і операції з ними, ця інформація знаходиться в JSON форматі і складається з списку сутностей типу ModelProtocol зображену на рис. 3.1

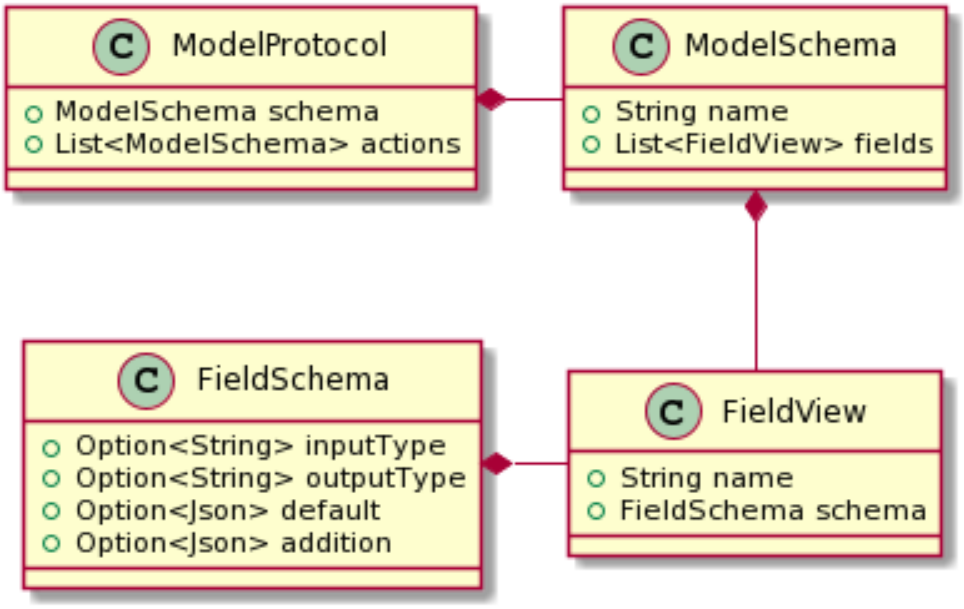


Рис. 3.1 Діаграма класів протоколу

Як видно з діаграми класів кожна модель розповідає про свою схему та схему можливих дій над нею, при цьому ModelSchema складається з імені і списку FieldView, що зберігають ім’я поля та схему поля, яка в свою чергу має опціональні параметри типу вводу, виводу, стандартного значення та додаткової інформації, яка може знадобитись при роботі з цим типом поля.

Після ознайомлення з протоколом спілкування сервера з клієнтом потрібно перечислити список наявних полів, що підтримуються клієнтською частиною застосунку. Інформацію про наявні типи вводу та виводу наведено в таблиці 3.1

Таблиця 3.1

Тип поля	Значення за замовчуванням	Додаткова інформація	Опис
1	2	3	4
int	0	-	Поле цілочисельного типу
double	0.0	-	Поле типу числа з плаваючою комою
string	“”	-	Текстове поле
json	null	-	Тип поля для роботи з форматом JSON, включає в себе як і зручний перегляд так і повноцінний JSON редактор
bool	false	-	Поле умовного типу
wysiwyg	“”	-	Поле текстового типу з багатим редактором контенту, що дає змогу записувати візуальну інтерпритацію контенту у вигляді HTML без потребу користувачу знати його.
date-time	Час старту СКВ	-	Тип поля для роботи з часом. Дає змогу як переглядати час так і редагувати в календарі та на годиннику час і точну дату.

Таблиця 3.1 (закінчення)

1	2	3	4
list	Пустий список	Поле elementSchema, що включає інформацію про тип елементів списку	Поле типу списка, що може мати елементи довільного типу, що підтримується системою
enum	Перше значення переліку	Поле values, що інформує про всі наявні значення переліку	Поле, що описує перелік або єнумерацію, значення якої вже визначено і надаються на вибір користувачу
option	-	Поле elementSchema, що включає інформацію про тип опціонального значення	Поле, що описує необов’язкові елементи, тип яких може бути будь-яким, який підтримується системою
file	-	-	Поле, яке дає змогу управляти контентом у вигляді файла
url-image	-	-	Поле для роботи з картинками представленими в вигляді url
model	-	Поле relation, що вказує на тип реляції та ім’я моделі	Поле для вказання реляцій між моделями контенту

Оскільки розроблювана СКВ має бути безкоштовним продуктом з відкритим кодом, то спочатку було створено репозиторій на ресурсі github. Далі було сформовано структуру проекту, де в теці admin-ui лежить клієнтська частина СКВ, а серверні реалізації для конкретних мов лежать у відповідних теках. Оскільки в рамках дипломної роботи буде реалізована серверна імплементація лише для однієї мови, а саме Scala – то так тека, що включає в себе серверну реалізацію для Scala і буде називатись. Також у корені проекта є файл .gitignore в якому прописано файли версіонуванням якиї не потрібно займатись git, а саме: вихідний застосунок, проміжні результати, файли редакторів та IDE. В корені проекту знаходяться файли LICENSE та Readme.md в яких знаходиться ліцензія проекту та короткий опис відповідно. Важливим файлом тут є .travis.yml , який відповідає за автоматизацію збірки і випуску СКВ. Кожен раз коли на github створюється Pull Request для внесення змін Travis автоматично запускає компіляцію та прогон тестів СКВ, а коли цей Pull Request зіллють з головною віткою, то крім цих етапів СКВ також отримає нову версію, що потрапить у Maven Repository. Тонкощі реалізації серверної та клієнтської частини буде наведено далі.

3.2 Серверна частина

Для роботи з серверною частиною було створено sbt проект і написано build.sbt файл, що описує структуру проекту, включаючи автоматичну відправку готових версій СКВ в центральне сховище бібліотек під назвою Maven Repository. Також для СКВ було створено документацію за допомогою інструменту під назвою Docusaurus та відповідного плагіну для sbt. Документація написана мовою розмітки Markdown і знаходиться в теці docs та решта сайту документації в теці website. Документація автоматично потрапляє на так звані github pages і її можна одразу переглянути.

Крім вказаних конфігурацій, версій залежностей і інформації про паблішинг нових версій СКВ, важливим є те, що певні модулі мають

додатковий етап збірки, а саме це модулі admin-akka, admin-http4s. Цей додатковий етап збірки запускає збірку клієнтської частини і додає готовий додаток в ресурси кінцевого артефакту, таким чином за бажанням розробник може не створювати окремо сервер, де він розвертає клієнтську частину, а вона може автоматично віддаватись самим сервером шляхом віддачі ресурсів клієнтської частин.

Основними робочими одиницями в структурі серверної реалізації є класи ModelSchema, ModelService, ModelAdmin та AdminExtension. Решта ж або є допоміжними і створенні для зручності розробників або використовуються в наведених класах. Для кращого розуміння принципу роботи СКВ далі буде детально розібрано кожен з цих класів.

ModelSchema – це ключовий клас в реалізації СКВ, він не тільки як уже раніше було зазначено в пункті 3.1 відповідає за опис моделі поза межами сервера, а й використовується багатьма компонентами власне сервера. Важливим моментом є принцип створення самої схеми, адже це відбувається автоматично за допомогою метапрограмування. Принцип доволі простий розробник декларує неявні змінні для кожного типу полів наявних в моделі і таким чином система автоматично будує схему використовуючи неявні змінні в полі видимості. Це можна побачити на рис. 3.2

```
import magnolia.{debug, CaseClass, Magnolia, SealedTrait}

object ModelSchema {
  type Typeclass[T] = FieldSchema[T]

  @debug implicit def gen[T]: ModelSchema[T] = macro Magnolia.gen[T]

  def combine[T](ctx: CaseClass[FieldSchema, T]): ModelSchema[T] =
    ModelSchema(
      ctx.typeName.short,
      ctx.parameters.map(p =>
        FieldViewDto(
          name = p.label,
          schema = p.typeclass.toDto
        )
      )
    )
}
```

Рис. 3.2 Код генерації схеми моделі

Як видно з коду ми імпортуємо бібліотеку Magnolia, далі вказуємо, що для полів класів потрібно знаходити неявні параметри тайпкласу FieldSchema, обов''зково додається метод gen, який і генерує ModelSchema схему шляхом виклику макроса з бібліотеки. Також потрібно реалізувати метод combine, що описує як створюється схема. Для цього інтерфейс бібліотеки дає нам доступ до контексту класу для якого створюється схема(тип параметра CaseClass), далі ми беремо з контексту ім’я класу і проходимось по параметрам створюючи FieldView з відповідними іменами і тайпкласами FieldSchema. Якщо неявний параметр буде не знайдений, то завдяки директиві @debug на методі gen ми отримаємо детальну інформацію про це ще на етапі компіляції.

Далі розглянемо клас ModelService, який описує що потрібно робити в результаті виконання тієї чи іншої CRUD операції в СКВ. Декларація цього інтерфейсу зазначена на рис. 3.3

```scala
trait ModelService[F[_], Model, Id] {
  val genId: GenId[Id]
  val actions: List[Action[F, _]] = List.empty
  val searchActions: List[SearchAction[F, Model, _]] = List.empty
  def create(model: Model): F[Model]
  def upsert(model: Model): F[Model]
  def delete(ids: Seq[Id]): F[Unit]
  def find(page: PageRequest, dsl: SearchDSL): F[PageResponse[Model]]
}
```

Рис. 3.3 Код інтерфейсу ModelService

Як видно з декларації в інтерфейса generic тип ефекта(так званий Tagless Final підхід), моделі та її ідентифікатора і розробник повинен реалізувати генерацію ідентифікатора для своєї моделі при її створенні, CRUD операції та опціонально може передати користувацькі дії та користувацькі фільтри пошуку. З коробки наявний інтерфейс, який реалізує інтерфейс ModelService для будь якої моделі в пам’яті, що дуже зручно при тестуванні або коли потрібно швидко розробити MVP продукту без БД. Ця реалізація знаходиться в класі ModelService.Dummy (рис 3.4). Як уже зазначалось наявний модуль SlickModelService, який реалізує інтерфейс ModelService для реляційних БД з використанням бібліотеки Slick і в цій реалізації теж не обійшлось без макросів і метапрограмування.

```
class Dummy[F[_]: Sync, M, I: GenId](id: M => I) extends ModelService[F, M, I] {
  val genId: GenId[I] = GenId[I]
  val store: Ref[F, Map[I, M]] = Ref.unsafe[F, Map[I, M]](Map.empty)
  def create(model: M): F[M] = store.update(_ + (id(model) -> model)).as(model)
  def upsert(model: M): F[M] = store.update(_.updated(id(model), model)).as(model)
  def upsertMany(models: Seq[M]): F[Seq[M]] =
store.update(models.foldLeft(_)((s, m) => s.updated(id(m), m))).as(models)
  def delete(ids: Seq[I]): F[Unit] = store.update(_ -- ids).void
  def find(page: PageRequest): F[PageResponse[M]] =
    store.get
      .map(_.values.slice(page.offset, page.offset + page.size))
      .map(m => response(m.toSeq, m.size, page))
  def filter(query: M => Boolean): F[List[M]] = store.get.map(_.values.filter(query).toList)
  def find(query: M => Boolean): F[Option[M]] = store.get.map(_.values.find(query))
}
```

Рис. 3.4 Код реалізації зберігання контенту в пам’яті

Далі за списком клас ModelAdmin, що представляє собою агрегат інформації про конкретну модель, ModeService та ModelExtension, а також інші споміжні дані, саме тут зберігається протокол і HTTP інтерфейси взаємодії для цієї конкретної моделі.

Як результат кінцевим і найголовнішим класом є AdminExtension, бо саме цей клас агрегує всі інші існуючи модулі і об’єднує їх в готовий модуль СКВ. Він має допоміжні методи для додавання в нього HealthChecks(рис 3.5) і ModelAdmin, а також інших модулів.

```
trait HealthCheck[F[_]] {
  def isAlive: F[Boolean]
  def name: String
  def description: String
  def addition: Json = Json.Null
  def timeout: FiniteDuration = 1.seconds
}
```

Рис. 3.5 Сигнатура HealthCheck модуля перевірки стану сервера

Також AdminExtension формує кінцевий список HTTP інтерфейсів для цільового сервера. Оскільки він агрегує весь опис цих інтерфейсів, то він взмозі сформувати ці інтерфейси, проте писати цю логіку саме тут було б недоцільно, тому було створено два інтерфейси ToRoute та CorsMiddleware, що абстрактно потребують імплементувати CORS для інтерфейсів та як перетворити опис інтерфейсів на реалізацію, а згодом відповідні модуля для HTTP бібліотек імплементують ці інтерфейси(рис 3.6)

```
trait ToRoute[F[_], R] {
  def apply(endpoints: List[ServerEndpoint[_, _, _, Nothing, F]]): R
}
trait CorsMiddleware[F[_], R] {
  def apply(route: R): R
}
...
def routes[R]( implicit  toRoute: ToRoute[F, R],  corsMiddleware: CorsMiddleware[F, R]): R =
corsMiddleware(toRoute(endpoints))
```

Рис. 3.6 Інтерфейси ToRoute, CorsMiddleware та їх застосування в AdminExtension

Відповідно до визначеного функціоналу СКВ вона підтримує певний набір HTTP запитів кожен з яких потребує, щоб клієнт передавав JWT токен в Authorization заголовці HTTP запитів. Робота з токеном відбувається за допомогою кода на рис 3.7. Перелік цих запитів яких наведено у таблиці 3.2.

```
def encodeUser(user: User): String = {
    val claim = JwtClaim(content = user.asJson.noSpaces)
    JwtCirce.encode(claim, secret, JwtAlgorithm.HS256)
  }
def decodeUser(token: String): Option[User] =
  JwtCirce.decode(token, secret, Seq(JwtAlgorithm.HS256))
    .map(_.content).toOption.flatMap(parse(_).toOption).flatMap(_.as[User].toOption)
```

Рис. 3.7 Інтерфейси ToRoute, CorsMiddleware та їх застосув

Таблиця 3.2

Метод запиту HTTP	URL адреса	Параметри	Опис
1	2	3	4
GET	/admin/api/{ім'я моделі}/search	?size – кількість записів, що потрібно відображати на сторінці ?page – номер сторінки, що потрібно відобразити	Повертає пагінований список сутностей моделей.
POST	/admin/api/{ім'я моделі}/create	JSON представлення моделі, що має відповідати її схемі	Створити нову сутність моделі.
PUT	/admin/api/{ім'я моделі}/update	JSON представлення моделі, що має відповідати її схемі	Оновити існуючу сутність моделі.
DELETE	/admin/api/{ім'я моделі}/delete	JSON список ідентифікаторів сутностей, які потрібно видалити	Видалити сутності моделі
PUT	/admin/api/{ім'я моделі}/action	name – ім'я операції data – заповнена форма для виконання дії, що відповідає схемі дії	Виконати користувацьку дію над моделею відповідно до імені
POST	/admin/login	username - password -	Авторизація в системі
GET	/admin/protocol	-	Повернути протокол усіх моделей СКВ

Таблиця 3.2 (закінчення)

1	2	3	4
GET	/health	-	Повертає інформацію про життєздатність усіх компонентів серверу та серверу вцілому
GET	/api/swagger	-	Повертає інформацію про документацію серверу в форматі OpenApi

3.3 Клієнтська частина

Для реалізації клієнтської частини застосунку спочатку було створено пустий React проект за допомогою пакетного менеджера npm та команди «npx create-react-app admin-ui», що створила базову структуру проекта з залежностями на React. Весь вміст клієнтської частини застосунку знаходиться в теці admin-ui і має стандартну структуру для такого проекта: тека public в якій знаходяться статичні файли, src – тут знаходяться власне JavaScript код, що описує клієнтську частину та файл package.json – є файлом конфігурації проекту і саме тут вказуються залежності застосунку та інші конфігурації. Далі було додано ряд додаткових залежностей, що є готовими рішенням для відомих стандартних задач при побудові клієнтського застосунку. Серед них варто виокремити залежність під назвою «react-router» саме завдяки ній у користувача буде можливість потрапляти на різні сторінки після натискання на відповідні кнопки в меню СКВ, тобто ця компонентна займається маршрутизацією і відображенням сторінок в нашому SPA застосунку відповідно до поточного адресу.

Кодова база проекту має деревовидну структуру, де на найвищому рівні знаходиться компонентна під назвою App і саме тут ініціалізуються всі інші компоненти(рис 3.8). Про реалізацію кожної з компонент і утилітарних класів буде написано далі в цьому розділі.

```
function App() {
  return (
    <MuiPickersUtilsProvider utils={DateFnsUtils}>
      <Router>
        <AuthProvider>
          <AdminToolbar />
          <Container component={Box} mt={5}>
            <Switch>
              <PrivateRoute exact path={['/', '/admin']}><AdminMain /></PrivateRoute>
              <Route path="/login"><AdminLogin /></Route>
              <PrivateRoute path="/model/:name"><AdminModel /> </PrivateRoute>
              <PrivateRoute path="/health"><AdminHealth /></PrivateRoute>
              <PrivateRoute path="/documentation"><AdminDocumentation /></PrivateRoute>
            </Switch>
            <ToastContainer />
          </Container>
        </AuthProvider>
      </Router>
    </MuiPickersUtilsProvider>
  );
}
```

Рис. 3.8 Код точки входу App.js

AuthProvider – це компонентна, що відповідає за всю роботу з користувачем і вона реалізована з використанням React Context API, тобто кожна інша компонетна, що є вкладеною в цю матиму доступ до її контекста, а саме це: токен, http клієнт з настроєним JWT токеном та базовим адресом сервера СКВ, інформація про поточного користувача, а також функції логіну, лог ауту, перевірки чи користувач має права доступу до тієї чи іншої сторінки, а також чи він авторизований. Тож ця компонетна знаходиться на найвищому

рівні і її контекст використовується в рамках інших для маніпуляцій над поточним користувачем (рис 3.9).

```
const { client, hasModelPermission, login, logout } = useAuth();
```

Рис. 3.9 Приклад використання AuthContext

Router – це компонетна з «react-router», що також знаходиться на найвищому рівні і також є контекстом для управління переходами між сторінкам застосунку, також дає доступ до інформації щодо поточної відкритої сторінки та історії переходів з можливістю змінювати її. Для цього всередині неї розміщено стандартну компоненту Switch, а всередині неї Route, а також написану додаткову компонентну PrivateRoute. Саме компонентни Route та PrivateRoute описують за яких умов користувач побачить вкладену в його тіло компонентну.

PrivateRoute – це невелика утиліта, що викликає функцію isAuther з контексту авторизації та дає потрапити на сторінку лише авторизованому користувачу, а інакше відправляє на сторінку авторизації.

HttpClient виконує роль абстракції над HTTP клієнтом з яким вміє взаємодіяти клієнтських застосунок, на ньому визначені методи для посилання тих чи інших HTTP запитів з розрахунком, що спілкування відбувається саме з сервером СКВ за визначеним протоколом і способом обробки помилок.

Утиліти Private та Guest – це дві сторони однієї медалі, вони використовують AuthContext і викликають функцію isAuthed, щоб зрозуміти відображати тільки авторизованому користувачу чи тільки гостю відповідно.

ToastContainer – утилітарна компонента, що відображає пуш нотифікації для користувача. Оскільки обрано пуш нотифікації в якості основного способу інформування користувача про події, що сталися в системі, то було використано бібліотеку «'react-toastify» і ToastContainer обов’язково має знаходитись на найвищому рівні для успішної роботи пуш нотифікацій. Для посилання таких нотифікацій достатньо будь де зробити імпорт об’єкта toast та викликати на

ньому відповідну функцію нотифікації, які до того ж можна гнучко конфігурувати.

AdminLogin виконує функцію авторизації користувача після введення ним ім’я та паролю за допомогою існуючої функції login з AuthContext.

AdminToolbar – відповідає за демонстрацію меню управління СКВ, що відображається лише, коли користувач авторизований і кожен пункт меню відображається чи ні в залежності від прав користувача для цього використовується вже наведена функція «hasPermission» з контексту авторизації. Саме тут знаходяться кнопки переходу між сторінками в СКВ, а саме перехід на: домашню сторінку, сторінку документації, здоров’я серверу, управління контентом, кнопки login та logout.

Для опису домашньої сторінки СКВ було створено компоненту AdminMain тут лише відображається коротка інформація про СКВ та посилання на документацію і її соціальні мережі.

В рамках логіки компоненти AdminHealth робиться запит на інтерфейс СКВ а адресою “/health” і далі в таблиці відображає детальна інформація про здоров’я сервісу.

AdminDocumentation представляє собою досить просту компоненту, що лише делегує всю роботу на бібліотеку Redoc, що вміє демонструвати документацію в зручному для людини вигляді в випадку, якщо вона описана в форматі OpenApi за допомогою json чи yaml. Тобто робиться запит на відповідний інтерфейс СКВ з адресою «/api/swagger» і вона автоматично відображається.

І нарешті ми дійшли до найголовнішої компонентни клієнтської частини, а саме AdminModel, що відповідає за виконання всіх можливих операцій над контентом. При старті робиться запит до СКВ за адресою «/admin/protocol» і отриманий у відповідь протокол використовується для побудови інтерфейсу взаємодії. З контексту react-router береться параметр шляху, що вказує з якою моделлю бажає працювати користувач і в протоколі шукається опис цієї моделі.

Далі використовується бібліотека «material-table» , що здатна відображати таблиці відповідно до заданої схеми тому тут використовується протокол для ініціалізації стопців, далі по протоколу обираються правильні компоненти вводу і виводу, початкові значення полів для нової сутності. Ця таблиця здатна відображати пагінований контент для цього потрібно реалізувати функцію, що приймає в якості параметрів розмір сторінки та її номер і повертає самі дані, номер сторінки і загальну кількість існуючих елементів. Також саме тут описано як виконувати створення, оновлення та видалення даних, а також перевірка прав доступу до цього функціоналу. Зліва від таблиці знаходиться ModelForm компонент, що відповідає за відображення користувацьких дій над обраним чи всім контентом. Він форма вводу полів дії будується за тим же прицином, що і для роботи з моделлю.

Для відображення компонентів вводу і виводу полів потрібно було спираючись на протокол відображати ті чи інші компоненти і цю проблему було вирішено за допомогою створення загальних компонет під назвою FieldInput та FieldOutput, що виконують роль контролера, який компонент вводу чи виводу відповідно потрібно відобразити. FieldInput чекає в якості параметрів схему поля, її геттер та сеттер, а FieldOutput лише схему та геттер. В обох випадках компоненти, які можна використовувати знаходяться в структурі Map з відповідними іменами і в залежності від схеми поля відображається обрана компонентна. На випадок, якщо вказано невідомий тип вводу чи виводу існує стандартний компонент, що повідомляє про це (рис 3.10).

```
function FieldInput({ field, item, setItem }) {
  const specificInput = inputs[field.schema.inputType] || DefaultInput;
  return specificInput({ field, item, setItem });
}

function FieldOutput({ field, item, setItem }) {
  const SpecificOutput = outputs[field.schema.outputType] || DefaultOutput;
  return <SpecificOutput field={field} item={item} setItem={setItem} />;
}
```

Рис. 3.10 Приклад використання AuthContext

ВИСНОВКИ ДО РОЗДІЛУ 3

В рамках цього розділу було описано реалізацію серверної та клієнтської частини системи управління контентом та тонкощі взаємодії між ними. Серверна частина написана мовою Scala з використанням sbt, tapir, http4s, akka http та magnolia для метапрограмування і формування схеми моделі. Клієнтська ж частина реалізована мовою JavaScript з використанням React та цілого ряду допоміжних технологій.

Важливим моментом розробки було влучно обрати протокол для спілкування клієнтської частини з серверною і як він буде формуватися на серверній частині застосунку. Оскільки серверна частина написана на Scala було обрано рішення написати макрос, що на етапі компіляції генерує схему для типів полів моделі, беручи implicit реалізації FieldSchema для цього типу і передає її далі в якості неявного параметру. Клієнтська частина в свою чергу будує інтерфейс взаємодії базуючих на схемі, що вона отримає від сервера СКВ.

В результаті імплементації було створено ефективну і легко розширювану клієнтську та серверну частини системи управління контентом, що відповідає всім раніше поставленим вимогам

РОЗДІЛ 4
ОГЛЯД РОБОТИ СТВОРЕНОЇ СИСТЕМИ УПРАВЛІННЯ КОНТЕНТОМ

При відкритті головної сторінки вперше з'явиться форму входу в СКВ (рис. 4.1)

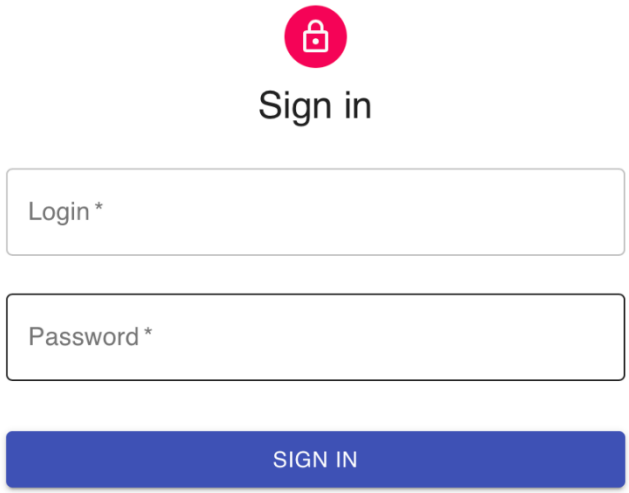


Рис. 4.1 Сторінка для входу в СКВ

Після успішного входу користувач потрапляє на головну сторінку (рис. 4.2) і в залежності від прав доступу користувача в меню СКВ він побачить ті чи інші опції.

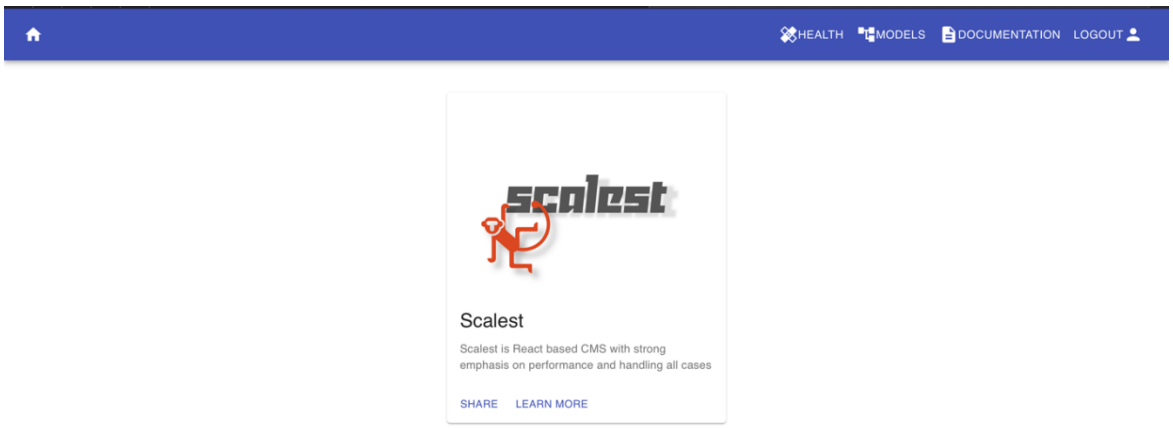


Рис. 4.2 Головна сторінка

Для перегляду здоров’я сервісу та його компонентів потрібно перейти в пункт меню “Health” (рис. 4.3)

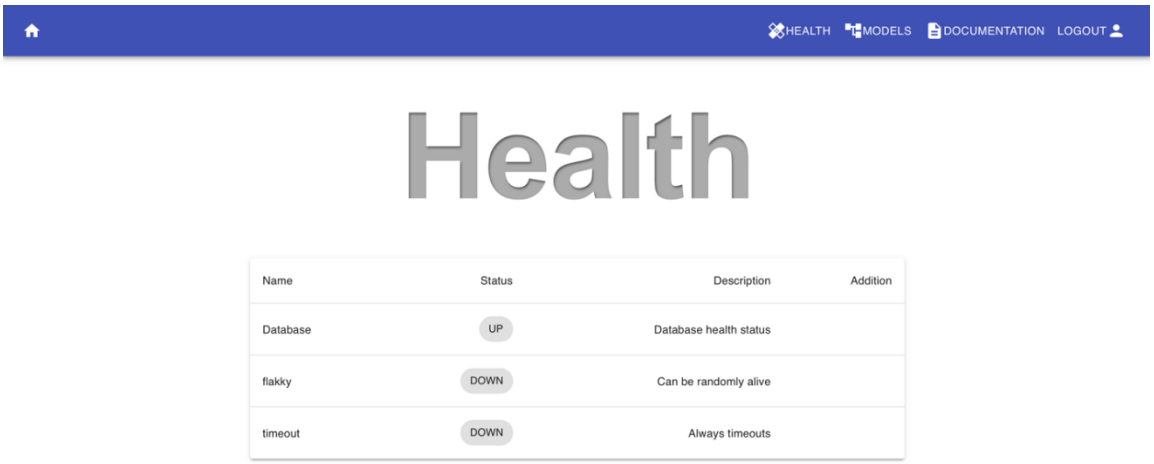


Рис. 4.3 Перегляд здоров’я

Якщо користувач бажає переглянути документацію сервіса та інтерфейсів СКВ – потрібно перейти в пункт меню Documentation. (рис. 4.4)

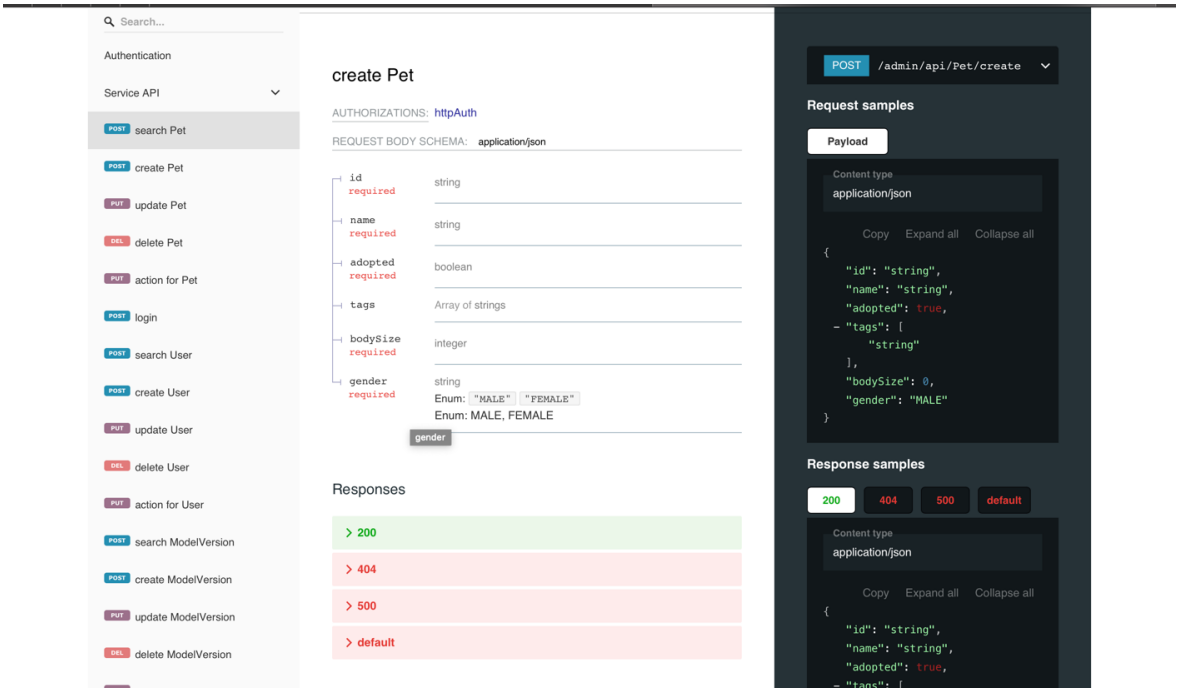


Рис. 4.4 Перегляд документації

Для перегляду моделей користувач має нажати пункт меню «Models» і обрати модель з якою бажає взаємодіяти після чого він потрапить на сторінку, що демонструє таблицю з існуючими сутностями (рис.4.5).

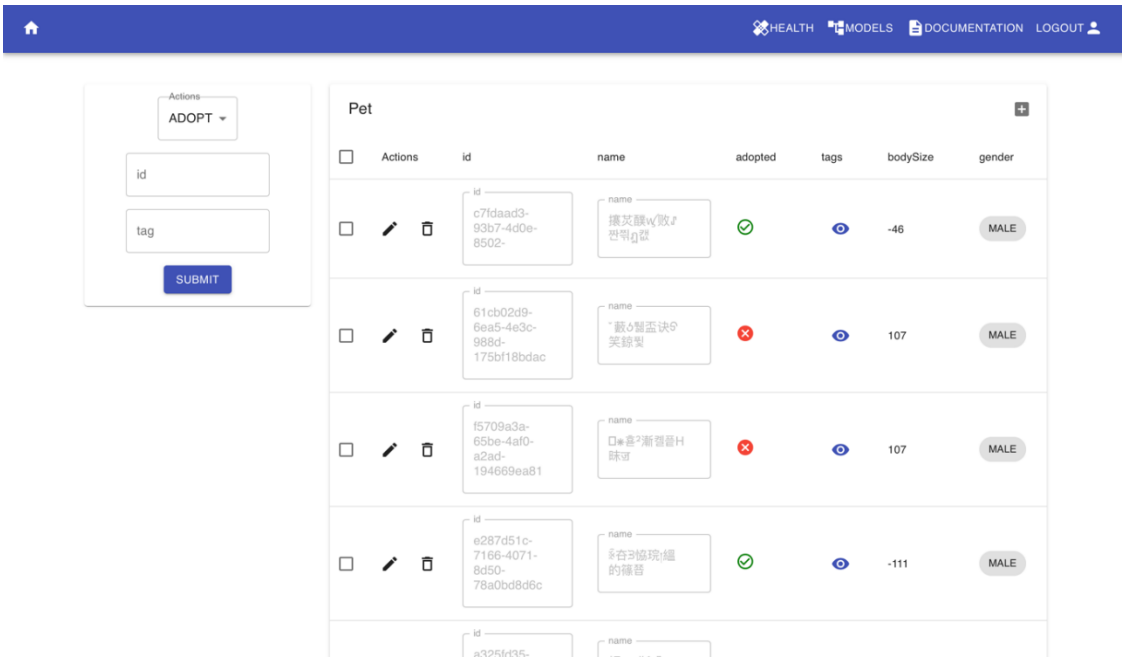


Рис. 4.5 Сторінка перегляду списку моделі

В цій же таблиці можна редагувати існуючі записи після натискання на кнопку редагування. Для збереження чи відміни результату знмі потрібно натиснути відповідну кнопку (рис.4.6).

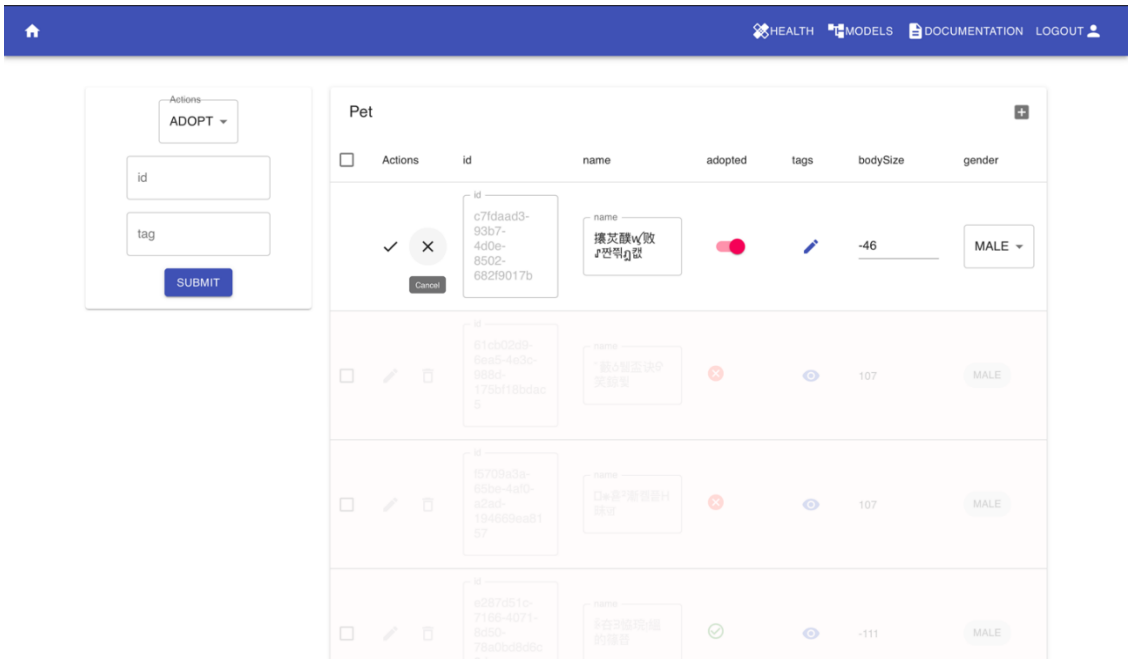


Рис. 4.6 Меню редагування моделі

Видалити сутність можна нажаттям на відповідну кнопку після чого потрібно підтвердити свій намір (рис.4.7).

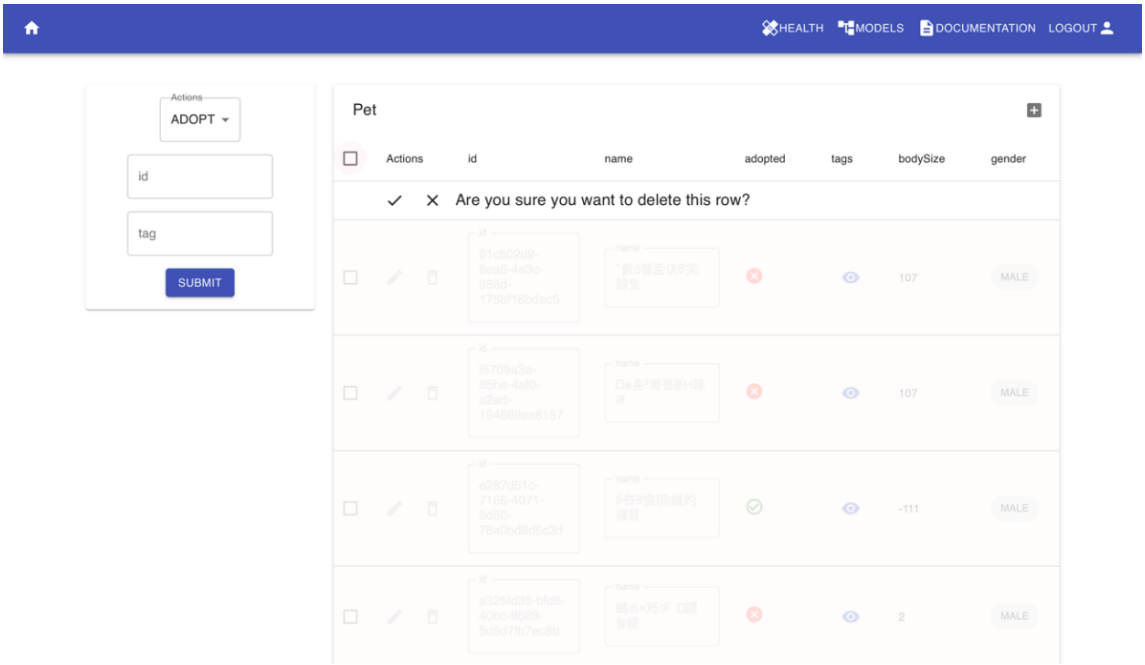


Рис. 4.7 Меню видалення моделі

Щоб створити нову сутність потрібно нажати на відповідний символ в правій стороні таблиці і ввести всі обов’язкові поля(рис.4.8).

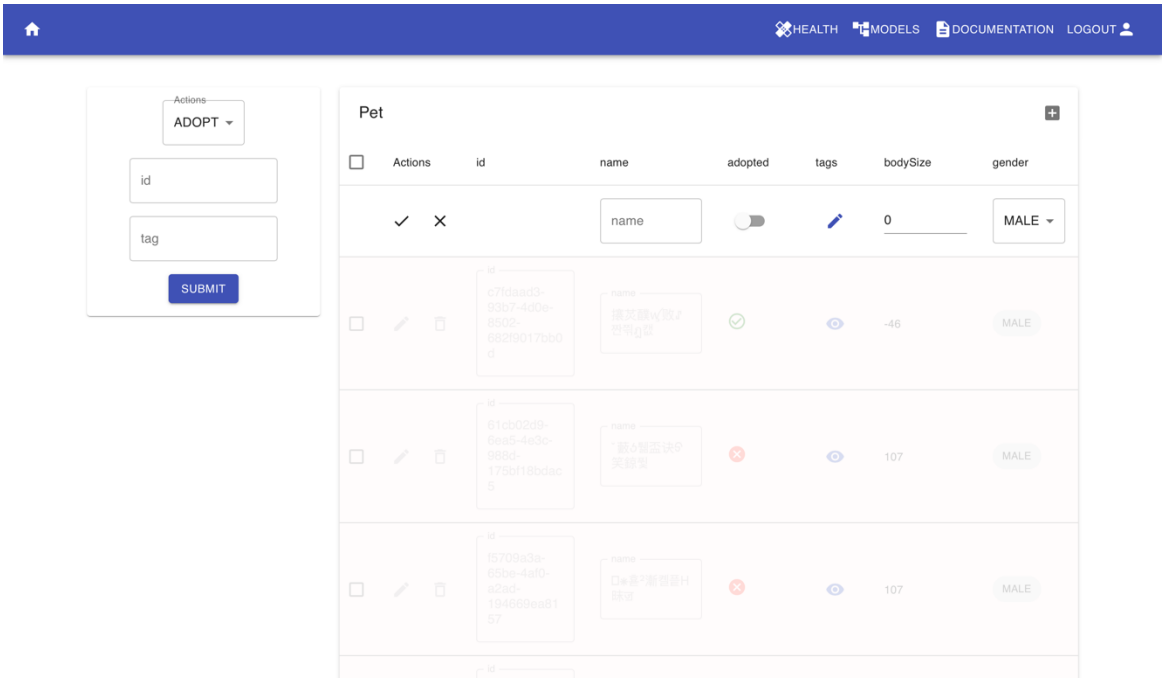


Рис. 4.8 Меню створення моделі

ВИСНОВКИ ДО РОЗДІЛУ 4

В цьому розділі представлено приклад використання створеної в рамках дипломної роботи системи управління контентом. Було розглянуто сценарії створення, видалення, зміни та перегляду контента, перегляду документації, входу в систему та перегляду стану здоров'я компонент сервера.

Як результат цей розділ демонструє, що реалізована в третьому розділі система має весь необхідний заявлений функціонал, має приємний та зрозумілий інтерфейс та низький час відповіді, що позитивно впливає на загальне враження від використання СКВ.

ВИСНОВКИ

Метою дипломного проекта є розробка системи управління контентом. Поява великої кількості однотипних в області взаємодії з контентом призвела до появи різноманітних систем управління контентом, що поділяються на Headless та класичні СКВ.

В першому розділі було розглянуто опис 5 існуючих СКВ: 2 з яких є Headless, 2 представляють собою Cloud CMS, а одна є класичною CMS. В процесі аналізу цих рішень було виявлено ряд спільних недоліків та запозичено влучні рішення з цих СКВ. Основною ціллю розробки було усунути ці недоліки та імплементувати влучні рішення, що уже існують в інших СКВ.

В другому розділі йдеться про обрані технології, бібліотеки та мови програмування для реалізації Headless CMS з клієнт-серверною архітектурою. Було обрано мову Scala для реалізації серверної частини через її багатий на функціонал компілятор та значний набір готових бібліотек під будь-яке завдання. Для реалізації клієнтської частини використовується JavaScript та фреймворк React і його Hook API, що дає змогу функціонально та зручно описувати компоненти та взаємодію між ними.

Третій розділ розкриває деталі реалізації системи управління контентом, можливі сценарії взаємодії з системою та описує протокол взаємодії клієнтської частини з серверною.

Заключний четвертий розділ наглядно демонструє зовнішній вигляд інтерфейсу клієнтської частини застосунку принцип роботи розробленої СКВ та її основний наявний функціонал.

Отже, описана дипломному проекті система управління контентом відповідає поставленим у першому розділі вимогам, вирішує недоліки існуючих рішень, включає в себе їх влучні властивості, а також має ряд нових властивостей, яких немає в існуючих СКВ. Як результат влучно спроектованої

системи існує чимало шляхів для розвитку даної системи, включаючи: реалізацію серверу СКВ для інших мов програмування, розширення підтримуваних типів полів, додавання готових рішень для популярних проблем.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Strapi [Електронний ресурс] – Режим доступу до ресурсу: https://strapi.io/
2. Keystone [Електронний ресурс] – Режим доступу до ресурсу: https://www.keystonejs.com/
3. Contenful [Електронний ресурс] – Режим доступу до ресурсу: https://www.contentful.com/
4. Wordpress [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wordpress.org/
5. Buttercms [Електронний ресурс] – Режим доступу до ресурсу: https://buttercms.com/
6. Functional Programming in Scala / P. Chiusano, R. Bjarnason. – Washington, Manning Publications, 2014. – 320 p
7. Tagless Final [Електронний ресурс] – Режим доступу до ресурсу: https://degoes.net/articles/tagless-horror
8. ZIO [Електронний ресурс] – Режим доступу до ресурсу: https://zio.dev/
9. Cats Effect [Електронний ресурс] – Режим доступу до ресурсу: https://typelevel.org/cats-effect/
10. Monix [Електронний ресурс] – Режим доступу до ресурсу: https://monix.io/
11. Tapir [Електронний ресурс] – Режим доступу до ресурсу: https://tapir.softwaremill.com/
12. Http4s [Електронний ресурс] – Режим доступу до ресурсу: https://http4s.org/
13. Akka Http [Електронний ресурс] – Режим доступу до ресурсу: https://doc.akka.io/docs/akka-http/current/index.html

14. Programming Scala, 2nd Edition / D. Wampler, A. Payne – O'Reilly Media, 2014 – 583 p
15. Slick [Електронний ресурс] – Режим доступу до ресурсу: http://scala-slick.org/
16. Learning Concurrent Programming in Scala / A. Prokopec – Packt Publishing, 2017. – 368 p
17. React [Електронний ресурс] – Режим доступу до ресурсу: https://uk.reactjs.org/

ДОДАТОК А

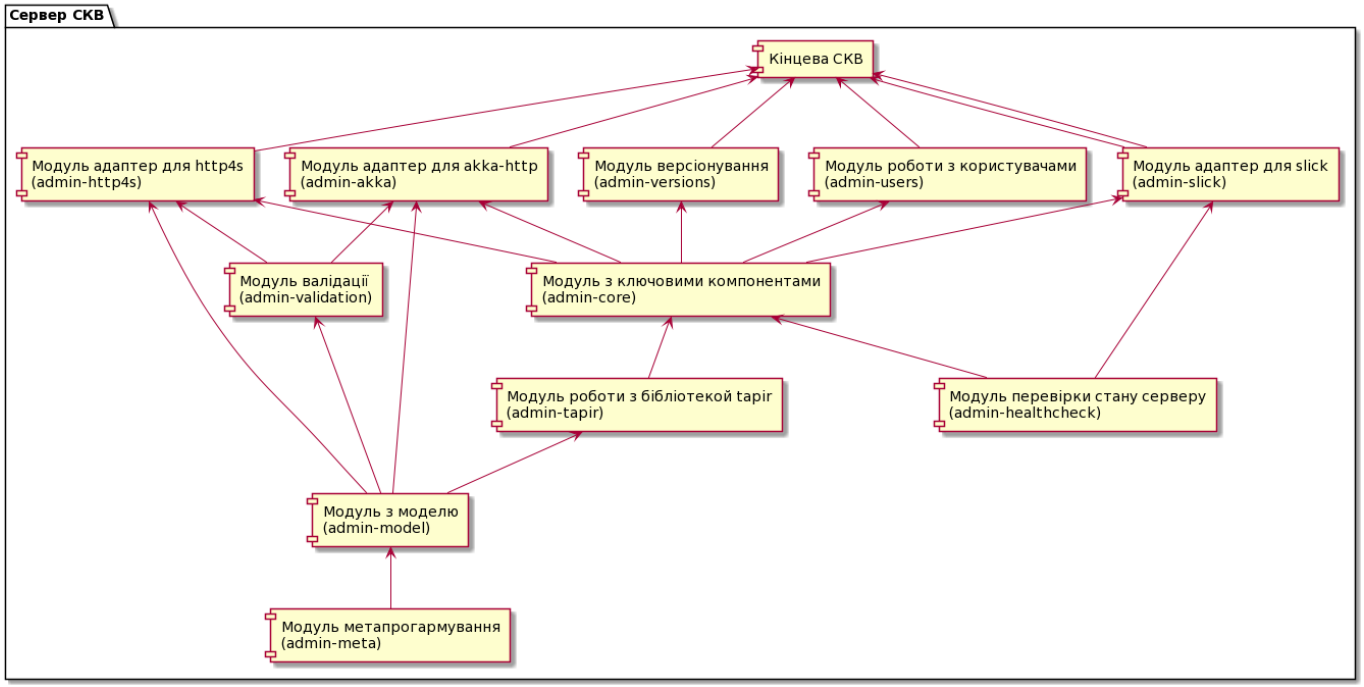
СИСТЕМА УПРАВЛІННЯ КОНТЕНТОМ

СХЕМА СТРУКТУРНА

Аркушів 1

Київ – 2020

ІАЛЦ.467200.



					ІАЛЦ 467200.004 Д1	
Зм.	Арк.	Прізвище	Підп	Дата		
Розроб.		*Дубинський О.І.*			*Система управління контентом*	*Літ.* / *Арк.* / *Аркушів*
Перевірив.		*Сімоненко В.П.*			*.Схема структурна*	1 / 1
						НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, кафедра ОТ гр. ІП-62
Н. кон.		*Сімоненко В.П.*				
Затв.		*Стіренко С.Г.*				

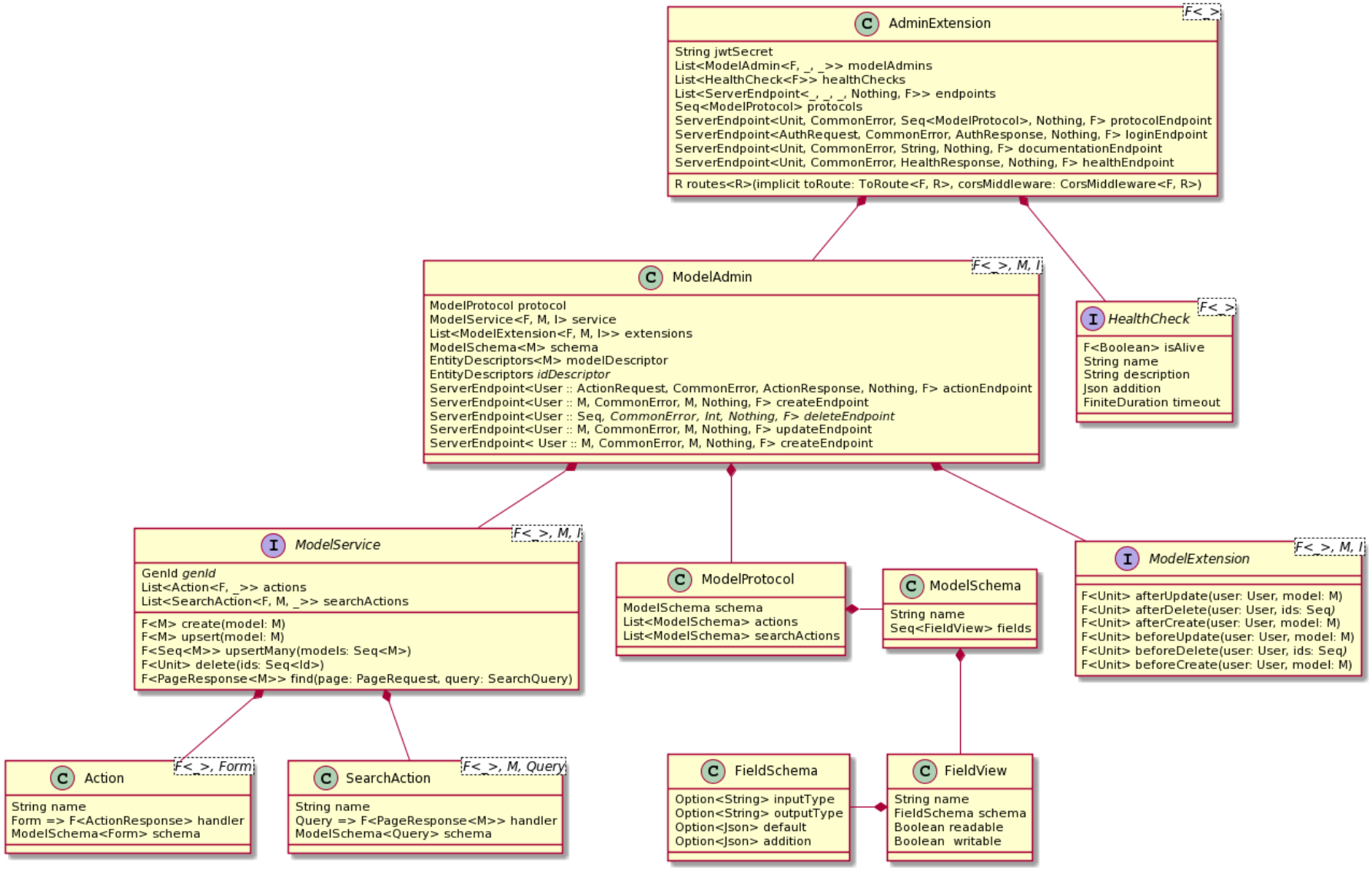
ДОДАТОК Б

СИСТЕМА УПРАВЛІННЯ КОНТЕНТОМ

ДІАГРАМА КЛАСІВ

Аркушів 1

Київ – 2020



Зм.	Арк.	Прізвище	Підп	Дата
Розроб.		*Дубинський О.І.*		
Перевірив.		*Сімоненко В.П.*		
Н. кон.		*Сімоненко В.П.*		
Затв.		*Стіренко С.Г.*		

ІАЛЦ 467200.005 Д2

Система управління контентом

.Схема функціональна

Літ.	*Арк.*	*Аркушів*
	1	1

НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, кафедра ОТ гр. ІП-62

ДОДАТОК В

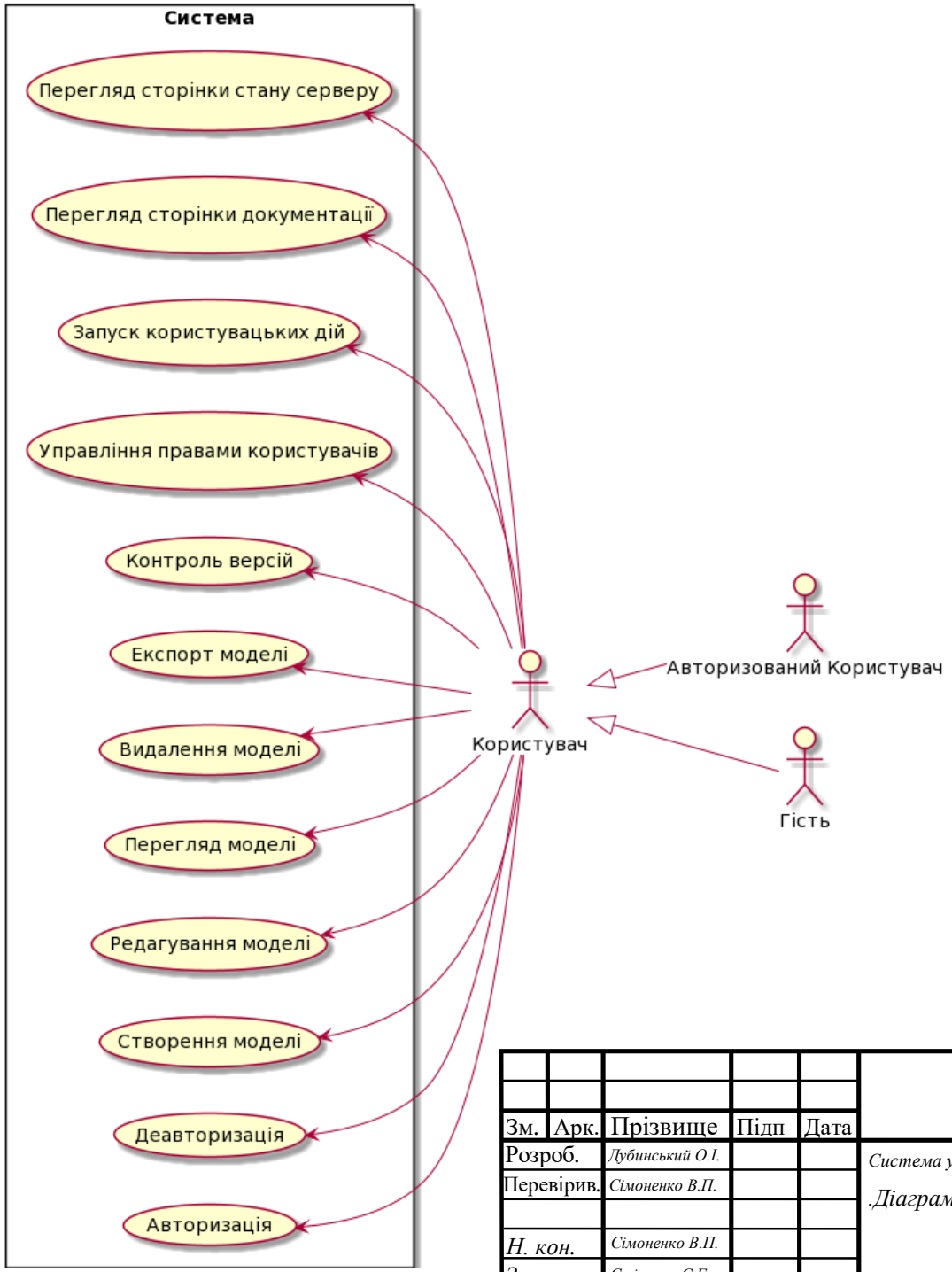
СИСТЕМА УПРАВЛІННЯ КОНТЕНТОМ

ДІАГРАМА ПРЕЦЕДЕНТІВ

Аркушів 1

Київ – 2020

ІАЛЦ.467200.



					ІАЛЦ 467200.006 Д3
Зм.	Арк.	Прізвище	Підп	Дата	
Розроб.		*Дубинський О.І.*			*Система управління контентом*
Перевірив.		*Сімоненко В.П.*			*.Діаграма прецедентів*
Н. кон.		*Сімоненко В.П.*			
Затв.		*Стіренко С.Г.*			

Літ.	*Арк.*	*Аркушів*
	1	1

НТУУ «КПІ ім. Ігоря Сікорського», ФІОТ, кафедра ОТ гр. ІП-62

ДОДАТОК Г

СИСТЕМА УПРАВЛІННЯ КОНТЕНТОМ

ЛІСТИНГ ПРОГРАМИ

Аркушів 41

Київ – 2020

App.js

```
import React from 'react';
import { Container, Box } from '@material-ui/core';
import { ToastContainer } from 'react-toastify';
import { BrowserRouter as Router, Switch, Route } from 'react-router-dom';
import { MuiPickersUtilsProvider } from '@material-ui/pickers';
import DateFnsUtils from '@date-io/date-fns';
import AdminToolbar from './component/toolbar/AdminToolbar';
import AdminHealth from './component/AdminHealth';
import AdminModel from './component/AdminModel';
import AdminLogin from './component/AdminLogin';
import AdminMain from './component/AdminMain';
import AdminDocumentation from './component/AdminDocumentation';
import PrivateRoute from './util/PrivateRoute';
import { AuthProvider } from './context/AuthContext';
import 'react-toastify/dist/ReactToastify.css';

function App() {
  return (
    <MuiPickersUtilsProvider utils={DateFnsUtils}>
      <Router>
        <AuthProvider>
          <AdminToolbar />
          <Container component={Box} mt={5}>
            <Switch>
              <PrivateRoute exact path={['/', '/admin']}>
                <AdminMain />
              </PrivateRoute>
              <Route path="/login">
                <AdminLogin />
              </Route>
              <PrivateRoute path="/model/:name">
                <AdminModel />
              </PrivateRoute>
              <PrivateRoute path="/health">
```

```
            <AdminHealth />
          </PrivateRoute>
          <PrivateRoute path="/documentation">
            <AdminDocumentation />
          </PrivateRoute>
        </Switch>
        <ToastContainer />
      </Container>
    </AuthProvider>
  </Router>
 </MuiPickersUtilsProvider>
 );
}

export default App;
```

./context/AuthContext.js

```
import React, { useState } from 'react';
import HttpClient from '../util/HttpClient';

const AuthContext = React.createContext();

function parseJwt(token) {
 const base64Url = token.split('.')[1];
 const base64 = base64Url.replace(/-/g, '+').replace(/_/g, '/');
 const jsonPayload = decodeURIComponent(atob(base64).split('').map((c) =>
`%${(`00${c.charCodeAt(0).toString(16)}`).slice(-2)}`).join(''));

 return JSON.parse(jsonPayload);
}

function AuthProvider(props) {
 const [token, setToken] = useState(null); // localStorage.getItem('token')
 const isAuthed = () => token != null;
 const [user, setUser] = useState(isAuthed() ? parseJwt(token) : null);
```

```
  const [client, setClient] = useState(new HttpClient(isAuthed() ? { Authorization: `Bearer
${token}` } : {}));
  function hasPermission(permission) {
    if (!user) return false;
    if (user.isSuperUser) return true;
    return user.permissions.find((p) => p.$type === permission);
  }
  function hasModelPermission(model, permission) {
    if (!user) return false;
    if (user.isSuperUser) return true;
    return user.permissions.find((p) => p.$type === permission && p.model === model);
  }

  async function login(username, password) {
    const res = await client.post('/admin/login', { data: { username, password } });

    if (res.isOk) {
      const jwtToken = res.data.token;
      localStorage.setItem('token', jwtToken);
      setToken(jwtToken);
      setUser(parseJwt(jwtToken));
      setClient(new HttpClient({ Authorization: `Bearer ${jwtToken}` }));
    }

    return res;
  }

  const logout = () => {
    localStorage.removeItem('token');
    setUser(null);
    setToken(null);
    setClient(new HttpClient({}));
  };

  return (
```

```
    <AuthContext.Provider
      value={{
        token, client, user, hasPermission, hasModelPermission, login, logout, isAuthed,
      }}
      {...props}
    />
  );
}

function useAuth() {
  const context = React.useContext(AuthContext);
  if (context === undefined) {
    throw new Error('useAuth must be used within a AuthProvider');
  }
  return context;
}

export { AuthProvider, useAuth };
```

util/PrivateRoute.js

```
import React from 'react';
import {
  Route,
  Redirect,
} from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

function PrivateRoute({ children, ...rest }) {
  const { isAuthed } = useAuth();
  return (
    <Route
      {...rest}
      render={({ location }) => (isAuthed() ? (children) : (
        <Redirect
          to={{
            pathname: '/login',
```

```
            state: { from: location },
          }}
        />
      ))}
    />
  );
}

export default PrivateRoute;
```

util/Private.js

```
import { useAuth } from '../context/AuthContext';

function Private({ children }) {
  const { isAuthed } = useAuth();

  if (isAuthed()) return children;
  return null;
}

export default Private;
```

./util/Guest.js

```
import { useAuth } from '../context/AuthContext';

function Guest({ children }) {
  const { isAuthed } = useAuth();

  if (!isAuthed()) return children;
  return null;
}

export default Guest;
```

component/AdminHealth.js

```
import React, { useEffect, useState } from 'react';
import { makeStyles } from '@material-ui/core/styles';
import {
```

```
  Box, Paper, Typography, Table, TableBody, TableCell, TableContainer, TableHead, TableRow,
Grid,
} from '@material-ui/core';
import { useAuth } from '../context/AuthContext';
import EnumOutput from './schema/output/EnumOutput';

function AdminHealth() {
  const { client } = useAuth();
  const [health, setHealth] = useState();

  async function fetchData() {
    const settings = { logSuccess: true, logError: true };
    const response = await client.get('/health', settings);
    if (response.isOk) setHealth(response.data);
  }
  useEffect(() => { fetchData(); }, []);
  if (!health) return <span>Loading...</span>;

  return (
    <Grid container justify="center">
      <Grid container item xs={12} justify="center">
        <Box mb={5}>
          <Typography variant="h1" component="h2" className={classes.healthcheks}>
                    Health
          </Typography>
        </Box>
      </Grid>
      <Grid item xs={8}>
        <TableContainer component={Paper} elevation={2}>
          <Table className={classes.table} aria-label="simple table">
            <TableHead>
              <TableRow>
                <TableCell>Name</TableCell>
                <TableCell align="right">Status</TableCell>
                <TableCell align="right">Description</TableCell>
```

```
              <TableCell align="right">Addition</TableCell>
            </TableRow>
          </TableHead>
          <TableBody>
            {health.statuses.map((row) => (
              <TableRow key={row.name}>
                <TableCell component="th" scope="row">{row.name}</TableCell>
                <TableCell align="right">
                  <EnumOutput item={row.status} />
                </TableCell>
                <TableCell align="right">{row.description}</TableCell>
                <TableCell align="right">{row.addition}</TableCell>
              </TableRow>
            ))}
          </TableBody>
        </Table>
      </TableContainer>
    </Grid>
  </Grid>
 );
}

export default AdminHealth;
```

component/ModelForm.js

```
import React, { useState } from 'react';
import {
  Card, FormControl, InputLabel, Select, MenuItem, Box, Button, TextField, Grid,
} from '@material-ui/core';
import { useAuth } from '../context/AuthContext';

import fieldInput from './schema/input/FieldInput';

function defaultForSchema(schema) {
  const emptyForm = {};
  schema.fields.forEach((f) => {
```

```
    emptyForm[f.name] = f.schema.default;
  });
  return emptyForm;
}

function ActionsSelect({
  info, action, setForm, setAction,
}) {
  function setActionFromEvent(event) {
    const selectedAction = info.actions.find((a) => a.name === event.target.value);
    setForm(defaultForSchema(selectedAction));
    setAction(selectedAction);
  }

  return (
    <FormControl variant="outlined">
      <InputLabel id="actions">Actions</InputLabel>
      <Select
        labelId="actions"
        id="actions-select"
        value={action.name}
        onChange={setActionFromEvent}
      >
        {
           info.actions.map((a) => <MenuItem key={a.name}
value={a.name}>{a.name}</MenuItem>)
         }
      </Select>
    </FormControl>
  );
}

function ModelForm({ info, tableRef }) {
  const [action, setAction] = useState(info.actions[0]);
  const [form, setForm] = useState(defaultForSchema(info.actions[0]));
```

```
  const { client } = useAuth();

  async function handleSubmit(event) {
   event.preventDefault();
   const settings = { data: { name: action.name, data: form }, logSuccess: true, logError: true };
   await client.put(`/admin/api/${info.schema.name}/action`, settings);
   if (tableRef.current) tableRef.current.onQueryChange();
  }

  if (!action) return <div>Loading</div>;

  return (
   <Card>
    <Box m={2}>
     <form onSubmit={handleSubmit}>
      <Grid
       container
       direction="column"
       justify="center"
       alignItems="center"
       spacing={2}
      >
       <Grid item>
        <ActionsSelect info={info} action={action} setAction={setAction} setForm={setForm}
/>
       </Grid>
       { action.fields.map((f) => (
        fieldInput({
         field: f,
         item: form[f.name],
         setItem: (e) => {
          const updatedForm = { ...form };
          updatedForm[f.name] = e;
          setForm(updatedForm);
         },
```

```
            })
          )).map((i) => <Grid item>{i}</Grid>) }
          <Grid item>
            <Button type="submit" variant="contained" color="primary"> Submit </Button>
          </Grid>
        </Grid>
      </form>
    </Box>
  </Card>
 );
}

export default ModelForm;
```

component/AdminModel.js

```
import React, { useEffect, useState, useRef } from 'react';
import { useParams } from 'react-router-dom';
import MaterialTable, { MTableToolbar } from 'material-table';
import { DeleteOutline } from '@material-ui/icons';
import { Chip, Paper, Grid } from '@material-ui/core';
import { CopyToClipboard } from 'react-copy-to-clipboard';
import FieldOutput from './schema/output/FieldOutput';
import fieldInput from './schema/input/FieldInput';
import { useAuth } from '../context/AuthContext';
import tableIcons from '../util/TableIcons';
import ModelForm from './ModelForm';

function AdminModel() {
  const tableRef = useRef();
  const { name } = useParams();
  const { client, hasModelPermission } = useAuth();
  const [info, setInfo] = useState();
  const [headers, setHeaders] = useState();
  const [actions, setActions] = useState([]);

  async function fetchData() {
```

```
    const protocolData = (await client.get('/admin/protocol', { logSuccess: true, logError: true
})).data;
    const modelInfo = protocolData.find((i) => i.schema.name === name);
    setInfo(modelInfo);
    setHeaders(modelInfo.schema.fields.map((f) => ({
      title: f.name,
      field: f.name,
      initialEditValue: f.schema.default,
      emptyValue: f.schema.default,
      editable: f.name === 'id' ? 'never' : 'always',
      render: (rowData) => {
        const el = <FieldOutput field={f} item={rowData[f.name]} />;
        if (f.name === 'id') return <CopyToClipboard
text={rowData[f.name]}>{el}</CopyToClipboard>;
        return el;
      },
      editComponent: ({ value, onChange }) => {
        const item = value;
        if (item === undefined) onChange(f.schema.default);
        return fieldInput({ field: f, item, setItem: onChange });
      },
    })));
    if (tableRef.current) tableRef.current.onQueryChange();
  }

  useEffect(() => { fetchData(); }, [name]);

  const onRowAdd = (newData) => client.post(`/admin/api/${name}/create`, { data: newData,
logSuccess: true, logError: true });
  const onRowUpdate = (newData) => client.put(`/admin/api/${name}/update`, { data: newData,
logSuccess: true, logError: true });
  const onRowDelete = (oldData) => client.delete(`/admin/api/${name}/delete`, { data:
[oldData.id], logSuccess: true, logError: true });
  const editable = {};
  if (hasModelPermission(name, 'UpdatePermission')) editable.onRowUpdate = onRowUpdate;
```

```
  if (hasModelPermission(name, 'CreatePermission')) editable.onRowAdd = onRowAdd;
  if (hasModelPermission(name, 'DeletePermission')) editable.onRowDelete = onRowDelete;
  const hasActions = () => info && info.actions.length !== 0;

  return (
    <Grid container spacing={3}>
      {hasActions() && (
        <Grid item xs={3}>
          <ModelForm info={info} tableRef={tableRef} />
        </Grid>
      )}
      <Grid item xs={hasActions() ? 9 : 12}>
        <MaterialTable
          tableRef={tableRef}
          icons={tableIcons}
          title={name}
          columns={headers}
          options={{
            search: false,
            selection: true,
            addRowPosition: 'first',
          }}
          actions={actions}
          data={(query) => client
            .post(`/admin/api/${name}/search?page=${query.page}&size=${query.pageSize}`, { data:
{ id: null } })
            .then((r) => r.data)
            .then((data) => ({
              data: data.content,
              page: data.number,
              totalCount: data.totalElements,
            }))}
          editable={editable}
        />
      </Grid>
```

```
    </Grid>
  );
}

export default AdminModel;
```

component/AdminLogin.js

```
import React, { useState } from 'react';
import {
  Avatar, Button, Container, CssBaseline, TextField, Typography,
} from '@material-ui/core';
import LockOutlinedIcon from '@material-ui/icons/LockOutlined';
import { makeStyles } from '@material-ui/core/styles';
import { useHistory } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';
function AdminLogin() {
  const history = useHistory();
  const { login } = useAuth();
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  async function submit(event) {
    event.preventDefault();
    const result = await login(username, password);

    if (result.isOk) history.replace('/');
    // else "error"
  }

  return (
    <Container component="main" maxWidth="xs">
      <CssBaseline />
      <div className={classes.paper}>
        <Avatar className={classes.avatar}>
          <LockOutlinedIcon />
        </Avatar>
```

```
<Typography component="h1" variant="h5">
  Sign in
</Typography>
<form className={classes.form} noValidate>
  <TextField
    variant="outlined"
    margin="normal"
    value={username}
    onChange={(v) => setUsername(v.target.value)}
    required
    fullWidth
    id="login"
    label="Login"
    name="login"
    autoComplete="login"
    autoFocus
  />
  <TextField
    value={password}
    onChange={(v) => setPassword(v.target.value)}
    variant="outlined"
    margin="normal"
    required
    fullWidth
    name="password"
    label="Password"
    type="password"
    id="password"
    autoComplete="current-password"
  />
  <Button
    type="submit"
    fullWidth
    variant="contained"
    color="primary"
```

```
            className={classes.submit}
            onClick={submit}
          >
            Sign In
          </Button>
        </form>
      </div>
    </Container>
  );
}

export default AdminLogin;
```

component/schema/input/FieldInput.js

```
/* eslint-disable import/no-cycle */
import React from 'react';
import BoolInput from './BoolInput';
import StringInput from './StringInput';
import IntInput from './IntInput';
import DoubleInput from './DoubleInput';
import EnumInput from './EnumInput';
import JsonInput from './JsonInput';
import ListInput from './ListInput';
import DateTimeInput from './DateTimeInput';

const inputs = {
  'bool-input': BoolInput,
  'string-input': StringInput,
  'int-input': IntInput,
  'enum-input': EnumInput,
  'double-input': DoubleInput,
  'json-input': JsonInput,
  'list-input': ListInput,
  'date-time-input': DateTimeInput,
};
```

```
function DefaultInput() {
  return <span>No input for such type</span>;
}

function fieldInput({ field, item, setItem }) {
  const specificInput = inputs[field.schema.inputType] || DefaultInput;
  return specificInput({ field, item, setItem });
}

export default fieldInput;
```

component/schema/output/FieldOutput.js

```
/* eslint-disable import/no-cycle */
import React from 'react';
import BoolOutput from './BoolOutput';
import StringOutput from './StringOutput';
import IntOutput from './IntOutput';
import DoubleOutput from './DoubleOutput';
import EnumOutput from './EnumOutput';
import JsonOutput from './JsonOutput';
import ListOutput from './ListOutput';

const outputs = {
  'bool-output': BoolOutput,
  'string-output': StringOutput,
  'int-output': IntOutput,
  'enum-output': EnumOutput,
  'double-output': DoubleOutput,
  'list-output': ListOutput,
  'json-output': JsonOutput,
  'date-time-output': StringOutput,
};

function DefaultOutput() {
  return <span>No output for such type</span>;
}
```

```
function FieldOutput({ field, item, setItem }) {
  const SpecificOutput = outputs[field.schema.outputType] || DefaultOutput;
  return <SpecificOutput field={field} item={item} setItem={setItem} />;
}

export default FieldOutput;
```

component/AdminMain.js

```
import React from 'react';

import { makeStyles } from '@material-ui/core/styles';
import {
  Grid, Card, CardActionArea, CardActions, CardContent, CardMedia, Button, Typography,
} from '@material-ui/core';

function AdminMain() {
  const classes = useStyles();

  return (
    <Grid
      container
      direction="column-reverse"
      justify="space-between"
      alignItems="center"
    >
      <Card className={classes.card}>
        <CardActionArea>
          <CardMedia
            component="img"
            alt="Scalest Logo"
            height="250"
            image={`${process.env.REACT_APP_API_URL}/scalest_logo`}
            title="Scalest Logo"
          />
          <CardContent>
```

```
          <Typography gutterBottom variant="h5" component="h2">
                    Scalest
          </Typography>
          <Typography variant="body2" color="textSecondary" component="p">
            Scalest is React based CMS with strong emphasis on
            performance and handling all cases
          </Typography>
        </CardContent>
      </CardActionArea>
      <CardActions>
        <Button size="small" color="primary">
                  Share
        </Button>
        <Button size="small" color="primary">
                  Learn More
        </Button>
      </CardActions>
    </Card>
  </Grid>
 );
}

export default AdminMain;
```

component/AdminDocumentation.js

```
import React from 'react';
import { RedocStandalone } from 'redoc';

function AdminDocumentation() {
 const swaggerUrl = `${process.env.REACT_APP_API_URL}/api/swagger`;
 return (
  <RedocStandalone specUrl={swaggerUrl} />
 );
}

export default AdminDocumentation;
```

ModelSchema.scala

```
package scalest.admin.schema

import magnolia.{debug, CaseClass, Magnolia, SealedTrait}
import scalest.admin.dto._

import scala.language.experimental.macros
import io.scalaland.chimney.dsl.TransformerOps

case class ModelSchema[T](name: String, fields: Seq[FieldViewDto]) {
  def dto: ModelSchemaDto = this.transformInto[ModelSchemaDto]
}

object ModelSchema {
  def apply[T](implicit MS: ModelSchema[T]): ModelSchema[T] = MS
  type Typeclass[T] = FieldSchema[T]

  @debug implicit def gen[T]: ModelSchema[T] = macro Magnolia.gen[T]

  def combine[T](ctx: CaseClass[FieldSchema, T]): ModelSchema[T] =
    ModelSchema(
      ctx.typeName.short,
      ctx.parameters.map(p => FieldViewDto(name = p.label, schema = p.typeclass.toDto)),
    )
}
```

ModelService.scala

```
package scalest.service

import cats.effect.Sync
import cats.effect.concurrent.Ref
import cats.implicits.toFunctorOps
import scalest.admin.action.{Action, SearchAction}
import scalest.admin.pagination.PageResponse.response
import scalest.admin.pagination.{PageRequest, PageResponse}
```

```
trait ModelService[F[_], Model, Id] {
  val genId: GenId[Id]
  val actions: List[Action[F, _]] = List.empty
  val searchActions: List[SearchAction[F, Model, _]] = List.empty
  def create(model: Model): F[Model]
  def upsert(model: Model): F[Model]
  def upsertMany(models: Seq[Model]): F[Seq[Model]]
  def delete(ids: Seq[Id]): F[Unit]
  def search(page: PageRequest, dsl: SearchDsl): F[PageResponse[Model]]
}

object ModelService {

  class Dummy[F[_]: Sync, M, I: GenId](id: M => I) extends ModelService[F, M, I] {
    val genId: GenId[I] = GenId[I]
    val store: Ref[F, Map[I, M]] = Ref.unsafe[F, Map[I, M]](Map.empty)
    def create(model: M): F[M] = store.update(_ + (id(model) -> model)).as(model)
    def upsert(model: M): F[M] = store.update(_.updated(id(model), model)).as(model)
    def upsertMany(models: Seq[M]): F[Seq[M]] = store.update(models.foldLeft(_)((s, m) =>
s.updated(id(m), m))).as(models)
    def delete(ids: Seq[I]): F[Unit] = store.update(_ -- ids).void
    def search(page: PageRequest, dsl: SearchDsl): F[PageResponse[M]] =
      store.get
        .map(_.values.slice(page.offset, page.offset + page.size))
        .map(m => response(m.toSeq, m.size, page))
    def filter(query: M => Boolean): F[List[M]] = store.get.map(_.values.filter(query).toList)
    def find(query: M => Boolean): F[Option[M]] = store.get.map(_.values.find(query))
  }

}
```

VersionsModelService.scala

```
package scalest.admin.versions

import cats.effect.Sync
import scalest.service.{GenId, ModelService}
```

```
trait VersionsModelService[F[_]] extends ModelService[F, ModelVersion, String] {
  final override val genId: GenId[String] = VersionsModelService.genModelVersionId
}
object VersionsModelService {
  implicit val genModelVersionId: GenId[String] = GenId.genUUID
  class Dummy[F[_]: Sync] extends ModelService.Dummy[F, ModelVersion, String](_.id) with
VersionsModelService[F]
}
```

ModelVersion.scala

```
package scalest.admin.versions

import java.time.LocalDateTime

import io.circe.generic.semiauto.deriveCodec
import io.circe.{Codec, Json}
import scalest.admin.schema.ModelSchema
import scalest.admin.versions.ModelActions.ModelAction
import scalest.tapir._

object ModelActions extends Enumeration {
  type ModelAction = Value
  val Create: ModelAction = Value("Create")
  val Update: ModelAction = Value("Update")
  val Delete: ModelAction = Value("Delete")
  implicit val codec: Codec[ModelAction] = Codec.codecForEnumeration(this)
}

case class ModelVersion(
  id: String,
  model: String,
  action: ModelAction,
  content: Json,
  author: String,
  creationDate: LocalDateTime = LocalDateTime.now(),
```

```
)

object ModelVersion {
  implicit val codec: Codec[ModelVersion] = deriveCodec
  implicit val modelSchema: ModelSchema[ModelVersion] = ModelSchema.gen[ModelVersion]
  implicit val modelED: EntityDescriptors[ModelVersion] = EntityDescriptors.create
}
```

VersionsModelAdmin.scala

```
package scalest.admin.versions

import cats.ApplicativeError
import io.circe.Encoder
import scalest.admin.ModelAdmin
import scalest.admin.schema.ModelSchema
import scalest.auth.UserTokenCodec

final class VersionsModelAdmin[F[_]: ApplicativeError[*[_], Throwable]](repo:
VersionsModelService[F])(
  implicit UTC: UserTokenCodec,
) extends ModelAdmin[F, ModelVersion, String](repo, List.empty) {
  def extension[M: ModelSchema: Encoder, I: Encoder]: VersionsExtension[F, M, I] =
VersionsExtension[F, M, I](repo)
}
```

VersionsExtension.scala

```
package scalest.admin.versions

import cats.ApplicativeError
import cats.implicits.toFunctorOps
import io.circe.Encoder
import io.circe.syntax._
import scalest.admin.ModelExtension
import scalest.admin.schema.ModelSchema
import scalest.auth.User
import scalest.service.{GenId, ModelService}
```

```
case class VersionsExtension[
  F[_]: ApplicativeError[*[_], Throwable],
  M: ModelSchema: Encoder,
  I: Encoder,
](repo: ModelService[F, ModelVersion, String])
    extends ModelExtension[F, M, I] {
  private val modelName: String = ModelSchema[M].name
  override def afterUpdate(user: User, model: M): F[Unit] =
    repo.create(ModelVersion(GenId.genUUID.gen, modelName, ModelActions.Update,
model.asJson, user.id)).void

  override def afterDelete(user: User, ids: Seq[I]): F[Unit] =
    repo.create(ModelVersion(GenId.genUUID.gen, modelName, ModelActions.Delete, ids.asJson,
user.id)).void

  override def afterCreate(user: User, model: M): F[Unit] =
    repo.create(ModelVersion(GenId.genUUID.gen, modelName, ModelActions.Create,
model.asJson, user.id)).void

}
```

health/package.scala

```
package scalest

import io.circe._
import io.circe.generic.semiauto._
import scalest.health.Statuses.Status

import scala.concurrent.Future
import scala.concurrent.duration._

package object health {

  trait HealthCheck[F[_]] {
    def isAlive: F[Boolean]
    def name: String
```

```
    def description: String
    def addition: Json = Json.Null
    def timeout: FiniteDuration = 1.seconds
  }

  object Statuses extends Enumeration {
    type Status = Value
    val Up: Status = Value("UP")
    val Down: Status = Value("DOWN")

    def fromBoolean(state: Boolean): Status = if (state) Up else Down

    implicit val codec: Codec[Status] = Codec.codecForEnumeration(this)
  }

  case class HealthCheckStatus(name: String, status: Status, description: String, addition: Json)

  object HealthCheckStatus {
    implicit val codec: Codec[HealthCheckStatus] = deriveCodec
  }

  case class HealthResponse(status: Status, statuses: Seq[HealthCheckStatus])

  object HealthResponse {
    implicit val codec: Codec[HealthResponse] = deriveCodec
  }

}
```

UserTokenCodec.scala

```
package scalest.auth

import io.circe.parser.parse
import io.circe.syntax._
import pdi.jwt.{JwtAlgorithm, JwtCirce, JwtClaim}
```

```
trait UserTokenCodec {
  def decodeUser(token: String): Option[User]
  def encodeUser(user: User): String
}

object UserTokenCodec {
  case class JwtSecretCodec(secret: String) extends UserTokenCodec {
    def encodeUser(user: User): String = {
      val claim = JwtClaim(content = user.asJson.noSpaces)
      JwtCirce.encode(claim, secret, JwtAlgorithm.HS256)
    }

    def decodeUser(token: String): Option[User] =
      JwtCirce
        .decode(token, secret, Seq(JwtAlgorithm.HS256))
        .map(_.content)
        .toOption
        .flatMap(parse(_).toOption)
        .flatMap(_.as[User].toOption)
  }
}
```

ModelExtension.scala

```
package scalest.admin

import cats.Applicative
import cats.implicits.catsSyntaxApply
import scalest.auth.User

abstract class ModelExtension[F[_]: Applicative, M, I] { ext =>
  def afterUpdate(user: User, model: M): F[Unit] = Applicative[F].unit
  def afterDelete(user: User, ids: Seq[I]): F[Unit] = Applicative[F].unit
  def afterCreate(user: User, model: M): F[Unit] = Applicative[F].unit
  def beforeUpdate(user: User, model: M): F[Unit] = Applicative[F].unit
  def beforeDelete(user: User, ids: Seq[I]): F[Unit] = Applicative[F].unit
  def beforeCreate(user: User, model: M): F[Unit] = Applicative[F].unit
```

```
  def compose(other: ModelExtension[F, M, I]): ModelExtension[F, M, I] = new
ModelExtension[F, M, I] {
    override def afterUpdate(user: User, model: M): F[Unit] =
      ext.afterUpdate(user, model) *> other.afterUpdate(user, model)
    override def afterDelete(user: User, ids: Seq[I]): F[Unit] =
      ext.afterDelete(user, ids) *> other.afterDelete(user, ids)
    override def afterCreate(user: User, model: M): F[Unit] =
      ext.afterCreate(user, model) *> other.afterCreate(user, model)
    override def beforeUpdate(user: User, model: M): F[Unit] =
      ext.beforeUpdate(user, model) *> other.beforeUpdate(user, model)
    override def beforeDelete(user: User, ids: Seq[I]): F[Unit] =
      ext.beforeDelete(user, ids) *> other.beforeDelete(user, ids)
    override def beforeCreate(user: User, model: M): F[Unit] =
      ext.beforeCreate(user, model) *> other.beforeCreate(user, model)
  }
}

object ModelExtension {
  def empty[F[_]: Applicative, M, I]: ModelExtension[F, M, I] = new ModelExtension[F, M, I] {}
}
```

AdminExtension.scala

```
package scalest.admin

import cats.ApplicativeError
import com.typesafe.scalalogging.Logger
import scalest.health.HealthCheck
import sttp.tapir.server.ServerEndpoint

trait ToRoute[F[_], R] {
  def apply(endpoints: List[ServerEndpoint[_, _, _, Nothing, F]]): R
}

trait CorsMiddleware[F[_], R] {
  def apply(route: R): R
```

```
}

case class AdminExtension[F[_]](
  modules: List[TapirModule[F]] = List.empty,
)(implicit AE: ApplicativeError[F, Throwable]) { extension =>
  val logger: Logger = Logger("scalest.AdminExtension")
  val endpoints: List[ServerEndpoint[_, _, _, Nothing, F]] = modules.flatMap(_.endpoints)
  def routes[R](
    implicit
    toRoute: ToRoute[F, R],
    corsMiddleware: CorsMiddleware[F, R],
  ): R = corsMiddleware(toRoute(endpoints))

  //BUILDER METHODS
  def withModelAdmins(modelAdmins: ModelAdmin[F, _, _]*): AdminExtension[F] =
    withModules(modelAdmins :+ ProtocolModule(modelAdmins))
  def withHealthChecks(healthChecks: HealthCheck[F]*)(implicit T: Timeout[F]):
AdminExtension[F] =
    withModule(HealthModule[F](healthChecks))
  def withModule(module: TapirModule[F]): AdminExtension[F] = withModules(Seq(module))
  def withModules(modules: Seq[TapirModule[F]]): AdminExtension[F] = copy(modules =
extension.modules ++ modules)
}
```

ProtocolModule.scala

```
package scalest.admin

import cats.ApplicativeError
import cats.implicits._
import scalest.admin.dto._
import scalest.error
import scalest.tapir._
import sttp.tapir.server.ServerEndpoint

case class ProtocolModule[F[_]](modelAdmins: Seq[ModelAdmin[F, _, _]])(implicit C:
ApplicativeError[F, Throwable])
```

```
    extends TapirModule[F] {
  val protocols: Seq[ModelProtocol] = modelAdmins.map(_.protocol)
  val protocolEndpoint: ServerEndpoint[Unit, error.CommonError, Seq[ModelProtocol], Nothing,
F] = {
    commonEndpoint("protocol").get
      .in("admin" / "protocol")
      .out(jsonBody[Seq[ModelProtocol]])
      .tapir(_ => protocols.pure[F])
  }
  override val endpoints: List[ServerEndpoint[_, _, _, Nothing, F]] = List(protocolEndpoint)
}
```

TapirModule.scala

```
package scalest.admin

import sttp.tapir.server.ServerEndpoint

trait TapirModule[F[_]] {
  def endpoints: List[ServerEndpoint[_, _, _, Nothing, F]]
}
```

ModelAdmin.scala

```
package scalest.admin

import cats.ApplicativeError
import cats.implicits._
import io.circe.syntax._
import scalest.admin.action._
import scalest.admin.dto._
import scalest.admin.pagination.{PageRequest, PageResponse}
import scalest.admin.schema.ModelSchema
import scalest.auth._
import scalest.error
import scalest.exception.ApplicationException
import scalest.service.ModelService
import scalest.tapir._
import sttp.tapir.Codec.JsonCodec
```

```
import sttp.tapir.EndpointInput.Auth
import sttp.tapir.server.ServerEndpoint

import scala.language.implicitConversions

class ModelAdmin[F[_], M, I](
  service: ModelService[F, M, I],
  extensions: List[ModelExtension[F, M, I]],
)(
  implicit
  val MMS: ModelSchema[M],
  val MMD: EntityDescriptors[M],
  val IMD: EntityDescriptors[I],
  val UTC: UserTokenCodec,
  val F: ApplicativeError[F, Throwable],
) extends TapirModule[F] {
  val protocol: ModelProtocol = ModelProtocol(
    MMS.dto,
    service.actions.map(a => a.schema.dto.copy(name = a.name)),
    service.searchActions.map(a => a.schema.dto.copy(name = a.name)),
  )
  val userAuth: Auth.Http[User] =
    jwtAuth(
      UTC.decodeUser(_).toRight(ApplicationException("Cannot decode user")),
      UTC.encodeUser,
    )
  private val extension = extensions.foldLeft(ModelExtension.empty[F, M, I])(_.compose(_))
  private val searchLogic = (_: User, request: PageRequest) => service.search(request:
PageRequest)
  private val createLogic = (u: User, m: M) => service.create(m) <* extension.afterCreate(u, m)
  private val updateLogic = (u: User, m: M) => service.upsert(m) <* extension.afterUpdate(u, m)
  private val deleteLogic = (u: User, ids: Seq[I]) => service.delete(ids).as(1) <*
extension.afterDelete(u, ids)
  private val actionLogic = (_: User, r: ActionRequest) => {
    service.actions
```

```
      .find(_.name == r.name)
      .map(_.execute(r.data).getOrElse(F.pure(ActionFailure: ActionResponse)))
      .getOrElse(F.pure(ActionNotFound: ActionResponse))
  }

  private val searchEndpoint: ServerEndpoint[(User, PageRequest), error.CommonError,
PageResponse[M], Nothing, F] = {
    implicit val mpsjc: JsonCodec[PageResponse[M]] = MMD.pageResponseJsonCodec
    commonEndpoint(s"search ${MMS.name}").post
      .in(userAuth)
      .in("admin" / "api" / MMS.name / "search")
      .in(caseQuery[PageRequest])
      .out(jsonBody[PageResponse[M]])
      .tapir[F]((searchLogic.apply _).tupled)
  }

  private val createEndpoint: ServerEndpoint[(User, M), error.CommonError, M, Nothing, F] = {
    import IMD.e
    implicit val mjc: JsonCodec[M] = {
      val decoderWithGeneratedId =
        MMD.decoder.prepare(c => c.focus.map(_.mapObject(_.add("id",
service.genId.gen.asJson)).hcursor).getOrElse(c))
      MMD.copy(decoder = decoderWithGeneratedId)
    }.jsonCodec

    commonEndpoint(s"create ${MMS.name}").post
      .in(userAuth)
      .in("admin" / "api" / MMS.name / "create")
      .in(jsonBody[M])
      .out(jsonBody[M])
      .tapir[F]((createLogic.apply _).tupled)
  }

  private val updateEndpoint: ServerEndpoint[(User, M), error.CommonError, M, Nothing, F] = {
    implicit val mjc: JsonCodec[M] = MMD.jsonCodec
```

```
    commonEndpoint(s"update ${MMS.name}").put
      .in(userAuth)
      .in("admin" / "api" / MMS.name / "update")
      .in(jsonBody[M])
      .out(jsonBody[M])
      .tapir[F]((updateLogic.apply _).tupled)
  }

  private val deleteEndpoint: ServerEndpoint[(User, Seq[I]), error.CommonError, Int, Nothing, F]
= {
    implicit val ijc: JsonCodec[Seq[I]] = IMD.seqJsonCodec
    commonEndpoint(s"delete ${MMS.name}").delete
      .in(userAuth)
      .in("admin" / "api" / MMS.name / "delete")
      .in(jsonBody[Seq[I]])
      .out(jsonBody[Int])
      .tapir[F]((deleteLogic.apply _).tupled)
  }

  private val actionEndpoint: ServerEndpoint[(User, ActionRequest), error.CommonError,
ActionResponse, Nothing, F] = {
    commonEndpoint(s"action for ${MMS.name}").put
      .in(userAuth)
      .in("admin" / "api" / MMS.name / "action")
      .in(jsonBody[ActionRequest])
      .out(jsonBody[ActionResponse])
      .tapir[F]((actionLogic.apply _).tupled)
  }

  override def endpoints: List[ServerEndpoint[_, _, _, Nothing, F]] =
    List(
      searchEndpoint,
      createEndpoint,
      updateEndpoint,
      deleteEndpoint,
```

```
      actionEndpoint,
    )
}
```

SwaggerRoute.scala

```
package scalest.admin

import cats.ApplicativeError
import cats.implicits._
import io.circe.Printer
import io.circe.syntax._
import scalest.error
import scalest.tapir._
import sttp.tapir.openapi.{Info, OpenAPI}
import sttp.tapir.server.ServerEndpoint

object SwaggerRoute {

  def apply[F[_], R](
    info: Info = Info("Service API", "0.0.0"),
    documentedEndpoints: List[ServerEndpoint[_, _, _, _, F]],
  )(implicit C: ApplicativeError[F, Throwable], toRoute: ToRoute[F, R], cors: CorsMiddleware[F,
R]): R = {
    val printer: Printer = Printer.spaces2.copy(dropNullValues = true)
    val openApi: OpenAPI = documentedEndpoints.map(_.tag(info.title)).toOpenAPI(info)
    val yaml: String = openApi.toYaml
    val json: String = openApi.asJson.printWith(printer)
    def swaggerEndpoint(
      name: String,
      path: String,
      content: String,
    ): ServerEndpoint[Unit, error.CommonError, String, Nothing, F] =
      commonEndpoint(name)
        .in("api" / path)
        .out(stringBody)
        .tapir(_ => content.pure[F])
```

```
    val swaggerEndpoints = List(
      swaggerEndpoint("Swagger JSON", "swagger.json", json),
      swaggerEndpoint("Swagger YAML", "swagger.yaml", yaml),
      swaggerEndpoint("Swagger", "swagger", yaml),
    )

    cors(toRoute(swaggerEndpoints))
  }
}
```

HealthModule.scala

```
package scalest.admin

import cats.ApplicativeError
import cats.implicits._
import scalest.error
import scalest.health.{HealthCheck, HealthCheckStatus, HealthResponse, Statuses}
import scalest.tapir._
import sttp.tapir.server.ServerEndpoint

case class HealthModule[F[_]](healthChecks: Seq[HealthCheck[F]] = Seq.empty)(
  implicit timeout: Timeout[F],
  AE: ApplicativeError[F, Throwable],
) extends TapirModule[F] {
  private def check =
    healthChecks
      .map { h =>
        timeout
          .timeoutTo(h.isAlive, h.timeout, false.pure[F])
          .handleError(_ => false)
          .map(Statuses.fromBoolean)
          .map(status => HealthCheckStatus(h.name, status, h.description, h.addition))
      }
      .toList
      .sequence
      .map { statuses =>
```

```
      val status = statuses.map(_.status).find(_ == Statuses.Down).getOrElse(Statuses.Up)
      HealthResponse(status, statuses)
    }

  val healthEndpoint: ServerEndpoint[Unit, error.CommonError, HealthResponse, Nothing, F] = {
    commonEndpoint("health").get
      .in("health")
      .out(jsonBody[HealthResponse])
      .tapir(_ => check)
  }

  override val endpoints: List[ServerEndpoint[_, _, _, Nothing, F]] = List(healthEndpoint)
}
```

UsersModelService.scala

```
package scalest.admin.users

import cats.effect.Sync
import cats.implicits.{catsSyntaxApplicativeErrorId, catsSyntaxApplicativeId, toFlatMapOps}
import scalest.auth._
import scalest.exception.ApplicationException
import scalest.service.{GenId, ModelService}

trait UsersModelService[F[_]] extends ModelService[F, User, String] {
  def login(username: String, password: String): F[User]
}

object UsersModelService {
  implicit val genUserId: GenId[String] = GenId.genUUID

  class Dummy[F[_]: Sync] extends ModelService.Dummy[F, User, String](_.id) with
UsersModelService[F] {
    override def login(username: String, password: String): F[User] =
      if (username == AdminUsername && password == AdminPassword)
        User(genUserId.gen, AdminUsername, AdminPassword, List.empty, isSuperUser =
true).pure[F]
```

```
      else
        find(u => u.username == username && u.password == password)
          .flatMap(_.fold(ApplicationException(CredentialsIncorrect).raiseError[F, User])(_.pure[F]))
  }
}
```

UsersExtension.scala

```
package scalest.admin.users

import cats.syntax.applicativeError.catsSyntaxApplicativeErrorId
import cats.{Applicative, ApplicativeError}
import scalest.admin.ModelExtension
import scalest.admin.schema.ModelSchema
import scalest.auth.Permission._
import scalest.auth.User
import scalest.exception.ApplicationException

case class UsersExtension[F[_]: ApplicativeError[*[_], Throwable], M: ModelSchema, I](repo:
UsersModelService[F])
    extends ModelExtension[F, M, I] {
  private val name: String = ModelSchema[M].name
  def checkPermission(check: Boolean, error: String): F[Unit] =
    Applicative[F].whenA(check)(ApplicationException(error).raiseError[F, Unit])

  override def beforeUpdate(user: User, model: M): F[Unit] =
    checkPermission(user.hasPermission(UpdatePermission(name)), s"You don't have update
permission for $name")

  override def beforeDelete(user: User, ids: Seq[I]): F[Unit] =
    checkPermission(user.hasPermission(DeletePermission(name)), s"You don't have delete
permission for $name")

  override def beforeCreate(user: User, model: M): F[Unit] =
    checkPermission(user.hasPermission(CreatePermission(name)), s"You don't have create
permission for $name")
}
```

users/package.scala

```
package scalest.admin

import scalest.admin.schema.ModelSchema
import scalest.auth.User
import scalest.tapir.EntityDescriptors

package object users {
  implicit val modelSchema: ModelSchema[User] = ModelSchema.gen[User]
  implicit val modelED: EntityDescriptors[User] = EntityDescriptors.create
}
```

UsersModelAdmin.scala

```
package scalest.admin.users

import cats.ApplicativeError
import cats.implicits.toFunctorOps
import scalest.admin.{ModelAdmin, ModelExtension}
import scalest.auth.{AuthRequest, AuthResponse, User, UserTokenCodec}
import scalest.error
import scalest.tapir._
import sttp.tapir.server.ServerEndpoint

final class UsersModelAdmin[F[_]: ApplicativeError[*[_], Throwable]](
  service: UsersModelService[F],
  extensions: List[ModelExtension[F, User, String]],
)(
  implicit UTC: UserTokenCodec,
) extends ModelAdmin[F, User, String](service, extensions) {
  val loginEndpoint: ServerEndpoint[AuthRequest, error.CommonError, AuthResponse, Nothing,
F] =
    commonEndpoint("login").post
      .in("admin" / "login")
      .in(jsonBody[AuthRequest])
      .out(jsonBody[AuthResponse])
      .tapir { request =>
```

```
        service
          .login(request.username, request.password)
          .map(UTC.encodeUser)
          .map(AuthResponse.apply)
      }

  override def endpoints: List[ServerEndpoint[_, _, _, Nothing, F]] = List(loginEndpoint) ++
super.endpoints
}
```

PetModel.scala

```
package pet

import java.util.UUID

import io.circe.Codec
import io.circe.generic.JsonCodec
import pet.PetModel.Genders.Gender

object PetModel {
  def genUUID: String = UUID.randomUUID.toString

  object Genders extends Enumeration {
    type Gender = Value
    val Male = Value("MALE")
    val Female = Value("FEMALE")

    implicit val codec: Codec[Genders.Value] = Codec.codecForEnumeration(this)
  }

  @JsonCodec
  case class Pet(
    id: String,
    name: String,
    adopted: Boolean,
    tags: Seq[String],
```

```
    bodySize: Byte,
    gender: Gender,
  )
  object Pet {
    def tupled = (Pet.apply _).tupled
  }
}
```

Pets.scala

```
package pet

import io.circe.parser._
import io.circe.syntax._
import pet.PetModel.Genders.Gender
import pet.PetModel.{Genders, _}
import slick.ast.BaseTypedType
import slick.jdbc.H2Profile.api._
import slick.jdbc.JdbcType

object Pets {
  val pets = TableQuery[PetsTable]

  class PetsTable(tag: Tag) extends Table[Pet](tag, "pets") {

    val id = column[String]("id", O.PrimaryKey)

    val name = column[String]("name")

    val bodySize = column[Byte]("body_size")

    val tags = column[Seq[String]]("tags")

    val adopted = column[Boolean]("adopted")

    val gender = column[Gender]("gender")
```

```
    override def * = (id, name, adopted, tags, bodySize, gender).mapTo[Pet]
  }

  implicit val sexEnumMapper: JdbcType[Gender] with BaseTypedType[Gender] = {
    MappedColumnType.base[Gender, String](_.toString, Genders.withName)
  }

  implicit val seqStringMapper: JdbcType[Seq[String]] with BaseTypedType[Seq[String]] = {
    MappedColumnType.base[Seq[String], String](_.asJson.noSpaces,
parse(_).right.get.as[Seq[String]].right.get)
  }

}
```

PetHttp4sExample.scala

```
package pet

import cats.effect._
import cats.implicits._
import com.typesafe.config.{Config, ConfigFactory}
import org.http4s.HttpRoutes
import org.http4s.implicits._
import org.http4s.server.blaze.BlazeServerBuilder
import pet.PetHttp4sExample.config
import pet.PetModel.Pet
import scalest.admin.element.{ElementsModelAdmin, ElementsModelService}
import scalest.admin.http4s._
import scalest.admin.page.{PagesModelAdmin, PagesModelService}
import scalest.admin.slick.health._
import scalest.admin.users.{UsersModelAdmin, UsersModelService}
import scalest.admin.versions._
import scalest.admin.{AdminExtension, ModelAdmin, SwaggerRoute}
import scalest.auth.{User, UserTokenCodec}
import scalest.health.HealthCheck
import scalest.service.GenId
import slick.basic.DatabaseConfig
```

```
import slick.jdbc.H2Profile

import scala.concurrent.ExecutionContext.global
import scala.concurrent.duration._
import scala.concurrent.{Await, ExecutionContext}
import scala.util.Random

object PetHttp4sExample extends IOApp {
  val config: Config = ConfigFactory.load()
  implicit val genId: GenId[String] = GenId.genUUID
  implicit val userTokenCodec: UserTokenCodec = UserTokenCodec.JwtSecretCodec("secret")
  implicit val ec: ExecutionContext = global
  implicit val dc: DatabaseConfig[H2Profile] = DatabaseConfig.forConfig[H2Profile]("slick",
config)
  val petsService = new PetModelService[IO]
  val usersService = new UsersModelService.Dummy[IO]
  val versionsService = new VersionsModelService.Dummy[IO]
  val pagesService = new PagesModelService.Dummy[IO]
  val elementsService = new ElementsModelService.Dummy[IO]

  val pagesMA = new PagesModelAdmin[IO](pagesService, elementsService, List.empty)
  val elementsMA = new ElementsModelAdmin[IO](elementsService, List.empty)
  val versionsMA = new VersionsModelAdmin[IO](versionsService)
  val usersMA = new UsersModelAdmin(usersService, List(versionsMA.extension[User, String]))
  val petsMA = new ModelAdmin(petsService, List(versionsMA.extension[Pet, String]))

  val adminExtension: AdminExtension[IO] =
    AdminExtension[IO]()
      .withModelAdmins(petsMA, usersMA, versionsMA, pagesMA, elementsMA)
      .withHealthChecks(
        SlickHealthCheck(dc),
        HealthCheck.make(
          IO(Random.nextBoolean()),
          "flakky",
          "Can be randomly alive",
```

```
      ),
      HealthCheck.make(
        IO.sleep(2.seconds).as(true),
        "timeout",
        "Always timeouts",
      ),
    )

  val routes: HttpRoutes[IO] = adminExtension.routes <+>
      SwaggerRoute(documentedEndpoints = adminExtension.endpoints) <+>
      Http4sAdminUI() <+> pagesMA.routes

  Migration.migrate

  override def run(args: List[String]): IO[ExitCode] =
    BlazeServerBuilder[IO]
      .withNio2(true)
      .bindHttp(9090, "0.0.0.0")
      .withHttpApp(routes.orNotFound)
      .serve
      .compile
      .drain
      .as(ExitCode.Success)
}
```

PetModelService.scala

```
package pet

import cats.Applicative
import cats.implicits.catsSyntaxApply
import io.circe.Decoder
import io.circe.generic.semiauto.deriveDecoder
import pet.PetModel.Pet
import pet.PetModelService.AdoptForm
import pet.Pets.{pets, PetsTable}
import scalest.admin.action.{Action, ActionSuccess}
```

```
import scalest.admin.{action, FutureToEffect}
import scalest.service.GenId
import slick.basic.DatabaseConfig
import slick.jdbc.H2Profile

import scala.concurrent.ExecutionContext

class PetModelService[F[_]: FutureToEffect: Applicative](
  implicit val dc: DatabaseConfig[H2Profile],
  val ec: ExecutionContext,
) extends SlickModelService[F, Pet, String, PetsTable] {
  import dc.profile.api._
  override val genId: GenId[String] = GenId.genUUID
  override val tableQuery: TableQuery[PetsTable] = pets
  override val actions: List[action.Action[F, _]] = List(Adopt, Disadopt)
  override def idParam: IdParam = IdParam(_.id)
  //Actions
  def Adopt: Action[F, AdoptForm] =
    Action[F, AdoptForm]("ADOPT") { form =>
      update(form.id, _.copy(adopted = true, tags = Seq(form.tag))) *> ActionSuccess.pure[F]
    }
  def Disadopt: Action[F, AdoptForm] =
    Action[F, AdoptForm]("DISADOPT") { form =>
      update(form.id, _.copy(adopted = false, tags = Seq(form.tag))) *> ActionSuccess.pure[F]
    }
}
object PetModelService {
  case class AdoptForm(id: String, tag: String)

  object AdoptForm {
    implicit val decoder: Decoder[AdoptForm] = deriveDecoder[AdoptForm]
  }
  def apply[F[_]](implicit instance: PetModelService[F]): PetModelService[F] = instance
}
```